Software

IDC DOCUMENTATION

Distributed Application Control System (DACS)

Approved for public release;
distribution unlimited

Notice

This document was published June 2001 by the Monitoring Systems Operation of Science Applications International Corporation (SAIC) as part of the International Data Centre (IDC) Documentation. Every effort was made to ensure that the information in this document was accurate at the time of publication. However, information is subject to change.

Contributors

Lance Al-Rawi, Science Applications International Corporation
Warren Fox, Science Applications International Corporation
Jan Wüster, Science Applications International Corporation

Trademarks

BEA TUXEDO is a registered trademark of BEA Systems, Inc.
Isis is a trademark of Isis Distributed Systems.
Motif 2.1 is a registered trademark of The Open Group.
ORACLE is a registered trademark of Oracle Corporation.
SAIC is a trademark of Science Applications International Corporation.
Solaris is a registered trademark of Sun Microsystems.
SPARC is a registered trademark of Sun Microsystems.
SQL*Plus is a registered trademark of Oracle Corporation.
Sun is a registered trademark of Sun Microsystems.
Syntax is a Postscript font.
UltraSPARC is a registered trademark of Sun Microsystems.
UNIX is a registered trademark of UNIX System Labs, Inc.
X Window System is a registered trademark of The Open Group.

Ordering Information

The ordering number for this document is SAIC-01/3001.

This document is cited within other IDC documents as [IDC7.3.1].

Distributed Application Control System (DACS)

CONTENTS

About this Document i

- PURPOSE ii
- SCOPE ii
- AUDIENCE iii
- RELATED INFORMATION iii
- USING THIS DOCUMENT iii

Conventions v

Chapter 1: Overview 1

- INTRODUCTION 2
- FUNCTIONALITY 7
- IDENTIFICATION 9
- STATUS OF DEVELOPMENT 10
- BACKGROUND AND HISTORY 10
- OPERATING ENVIRONMENT 11

Hardware 11
Commercial-Off-The-Shelf Software 11

Chapter 2: Architectural Design 13

- CONCEPTUAL DESIGN 14
- DESIGN DECISIONS 18

Programming Language 18
Global Libraries 18
Database 19
Interprocess Communication (IPC) 19
Filesystem 20
UNIX Mail 20

FTP 20
Web 20
Design Model 21
Distribution and Backup Concept 23
Pipelines 25
Database Schema Overview 27
- **FUNCTIONAL DESCRIPTION** 28
Distributed Process Monitoring, Reliable Queueing, and Transactions 28
Data Monitoring 30
System Scheduling 30
Pipeline Processing 31
Workflow Monitoring 31
Automatic Processing Utilities 32
Operator Console 32
Interactive Processing 32
- **INTERFACE DESIGN** 34
Interface with Other IDC Systems 34
Interface with External Users 35
Interface with Operators 35

Chapter 3: Tuxedo Components and Concepts 37
- **PROCESSING UNITS** 38
- **TUXEDO COMPONENTS OF DACS** 38
Listener Daemons (tlisten, tagent) 38
Administrative Servers 42
Application Servers 43
IPC Resources 45
Special Files 45
Utility Programs 46

Chapter 4: Detailed Design 47
- **DATA FLOW MODEL** 48
- **PROCESSING UNITS** 54
Data Monitor Servers 54

scheduler/schedclient 77
tuxshell 83
dbserver, interval_router, and recycler_server 89
WorkFlow, SendMessage, and ProcessInterval 93
libipc, dman, and birdie 100
tuxpad, operate_admin, schedule_it, and msg_window 110
- DATABASE DESCRIPTION 119
Database Design 119
Database Schema 122

Chapter 5: Requirements 125
- INTRODUCTION 126
- GENERAL REQUIREMENTS 126
- FUNCTIONAL REQUIREMENTS 128
Availability Management 128
Message Passing 129
Workflow Management 131
System Monitoring 133
Reliability 134
- CSCI EXTERNAL INTERFACE REQUIREMENTS 137
- CSCI INTERNAL DATA REQUIREMENTS 142
- SYSTEM REQUIREMENTS 142
- REQUIREMENTS TRACEABILITY 144

References 175

Glossary G1

Index I1

Distributed Application Control System (DACS)

FIGURES

Figure 1.	IDC Software Configuration Hierarchy	3
Figure 2.	Relationship of DACS to Other Subsystems of IDC Software	4
Figure 3.	DACS Application for Automatic Processing	5
Figure 4.	DACS Application for Interactive Processing	7
Figure 5.	DACS as Middleware	8
Figure 6.	Conceptual Data Flow of the DACS for Automatic Processing	15
Figure 7.	Conceptual Data Flow of DACS for Interactive Processing	17
Figure 8.	Processing Requests from Message Queue	21
Figure 9.	Transaction in Detail	22
Figure 10.	Forwarding Agent	23
Figure 11.	Construction of a Pipeline	26
Figure 12.	Data Flow of the DACS for Automatic Processing	29
Figure 13.	Data Flow of the DACS for Interactive Processing	34
Figure 14.	Data Flow of DACS CSCs for Automatic Processing	50
Figure 15.	Control and Data Flow of DACS CSCs for Interactive Processing	53
Figure 16.	Data Monitor Context	55
Figure 17.	Data Monitor Acknowledgement to Scheduling System	56
Figure 18.	tis_server Data Flow	58
Figure 19.	Current Data and Skipped Interval Checks	60
Figure 20.	tiseg_server Data Flow	62
Figure 21.	ticron_server Data Flow	64
Figure 22.	tin_server Data Flow	66
Figure 23.	WaveGet_server Data Flow	68
Figure 24.	Scheduling System Data Flow	79
Figure 25.	tuxshell Data Flow	85

FIGURE 26.	DBSERVER DATA FLOW	89
FIGURE 27.	MONITORING UTILITY WORKFLOW	95
FIGURE 28.	WORKFLOW DATA FLOW	97
FIGURE 29.	TUXPAD DESIGN	112
FIGURE 30.	QINFO DESIGN	114
FIGURE 31.	SCHEDULE_IT DESIGN	115
FIGURE 32.	ENTITY RELATIONSHIP OF SAIC DACS CSCS	121
FIGURE 33.	DATA ARRIVAL EXAMPLE	139

Distributed Application Control System (DACS)

TABLES

TABLE I:	DATA FLOW SYMBOLS	v
TABLE II:	ENTITY-RELATIONSHIP SYMBOLS	vi
TABLE III:	TYPOGRAPHICAL CONVENTIONS	vii
TABLE IV:	TECHNICAL TERMS	vii
TABLE 1:	DATABASE TABLES USED BY DACS	27
TABLE 2:	MAP OF TUXEDO COMPONENTS TO SAIC DACS COMPONENTS	39
TABLE 3:	DACS/LIBIPC INTERVAL MESSAGE DEFINITION	103
TABLE 4:	LIBIPC API	106
TABLE 5:	DATABASE USAGE BY DACS	122
TABLE 6:	DACS OPERATIONAL MODES	127
TABLE 7:	FAILURE MODEL	136
TABLE 8:	TRACEABILITY OF GENERAL REQUIREMENTS	144
TABLE 9:	TRACEABILITY OF FUNCTIONAL REQUIREMENTS: AVAILABILITY MANAGEMENT	148
TABLE 10:	TRACEABILITY OF FUNCTIONAL REQUIREMENTS: MESSAGE PASSING	150
TABLE 11:	TRACEABILITY OF FUNCTIONAL REQUIREMENTS: WORKFLOW MANAGEMENT	153
TABLE 12:	TRACEABILITY OF FUNCTIONAL REQUIREMENTS: SYSTEM MONITORING	156
TABLE 13:	TRACEABILITY OF FUNCTIONAL REQUIREMENTS: RELIABILITY	158
TABLE 14:	TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS	161
TABLE 15:	TRACEABILITY OF CSCI INTERNAL DATA REQUIREMENTS	169
TABLE 16:	TRACEABILITY OF SYSTEM REQUIREMENTS	169

About this Document

This chapter describes the organization and content of the document and includes the following topics:

- Purpose
- Scope
- Audience
- Related Information
- Using this Document

About this Document

PURPOSE

This document describes the design and requirements of the Distributed Processing Computer Software Configuration Item (CSCI) of the International Data Centre (IDC). The collection of software is more commonly referred to as the Distributed Application Control System (DACS). The DACS consists of commercial-off-the-shelf (COTS) software and Science Applications International Corporation (SAIC) designed Computer Software Components (CSC) including server applications, client applications, one global library, and processing scripts.

SCOPE

The DACS software is identified as follows:

Title: Distributed Application Control System

Abbreviation: DACS

This document describes the architectural and detailed design of the software including its functionality, components, data structures, high-level interfaces, method of execution, and underlying hardware. Additionally, this document specifies the requirements of the software and its components. This information is modeled on the *Data Item Description for Software Design* [DOD94a] and *Data Item Description for Software Requirements Specification* [DOD94b].

AUDIENCE

This document is intended for all engineering and management staff concerned with the design and requirements of all IDC software in general and of the DACS in particular. The detailed descriptions are intended for programmers who will be developing, testing, or maintaining the DACS.

RELATED INFORMATION

See "References" on page 175 for a list of documents that supplement this document. The following UNIX Manual (man) Pages apply to the existing DACS software:

- *dbserver(1)*
- *dman(1)*
- *interval_router(1)*
- *libipc(3), birdie(1)*
- *recycler_server(1)*
- *schedclient(1), scheduler(1)*
- *SendMessage(1)*
- *tis_server(1), tiseg_server(1), ticron_server(1), tin_server(1), WaveGet_ server(1)*
- *tuxpad(1)*
- *tuxshell(1)*
- *WaveGet_server(1)*
- *WorkFlow(1)*

USING THIS DOCUMENT

This document is part of the overall documentation architecture for the IDC. It is part of the Software category, which describes the design of the software. This document is organized as follows:

- Chapter 1: Overview

  This chapter provides a high-level view of the DACS, including its functionality, components, background, status of development, and current operating environment.

- Chapter 2: Architectural Design

  This chapter describes the architectural design of the DACS, including its conceptual design, design decisions, functions, and interface design.

- Chapter 3: Tuxedo Components and Concepts

  This chapter describes key software components and concepts of the Transactions for UNIX Extended for Distributed Operations (Tuxedo) (COTS) software product used by the DACS.

- Chapter 4: Detailed Design

  This chapter describes the detailed design of the SAIC-supplied Distributed Processing CSCs, including their data flow, software units, and database design.

- Chapter 5: Requirements

  This chapter describes the general, functional, and system requirements of the DACS.

- References

  This section lists the sources cited in this document.

- Glossary

  This section defines the terms, abbreviations, and acronyms used in this document.

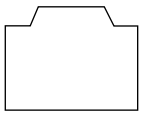
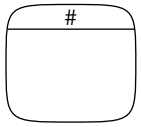
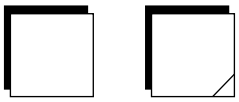
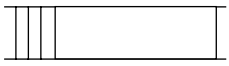
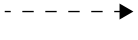
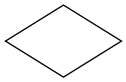
- Index

  This section lists topics and features provided in the document along with page numbers for reference.

Conventions

This document uses a variety of conventions, which are described in the following tables. Table I shows the conventions for data flow diagrams. Table II shows the conventions for entity-relationship diagrams. Table III lists typographical conventions. Table IV explains certain technical terms that are unique to the DACS and are used in this document. For convenience, these terms are also included in the Glossary, which is located at the end of this document.

TABLE I: DATA FLOW SYMBOLS

Description[1]	Symbol
host (computer)	
process	
external source or sink of data (left) duplicated external source or sink of data (right)	
data store (left), duplicated data store (right) D = disk store Db = database store MS = mass store	
queue	
control flow	
data flow	
decision	

1. Symbols in this table are based on Gane-Sarson conventions [Gan79].

TABLE II: ENTITY-RELATIONSHIP SYMBOLS

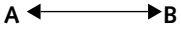
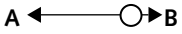
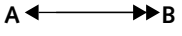
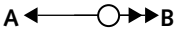
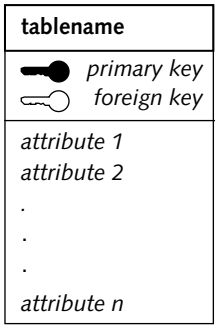
Description	Symbol
One **A** maps to one **B**.	
One **A** maps to zero or one **B.**	
One **A** maps to many **B**s.	
One **A** maps to zero or many **B**s.	
database table	

TABLE III: TYPOGRAPHICAL CONVENTIONS

Element	Font	Example
database table	**bold**	**interval**
database table and column, when written in the dot notation		**interval.***state*
database columns	*italics*	*state*
processes, software units, and libraries		*tuxshell*
user-defined arguments and variables used in parameter (par) files or program command lines		*target-interval-size*
COTS BEA/Tuxedo supplied titles of documents		*(DACS) Software User Manual* *Distributed Application Control System*
server software (all CAPS)		*BRIDGE*
computer code and output	`courier`	`interval_by_wfdisc()`
filenames, directories, and web sites		`/src/distributed/src/tis`
text that should be typed in exactly as shown		`man tis_server`

TABLE IV: TECHNICAL TERMS

Term	Description
admin server	Tuxedo server that provides interprocess communication and maintains the distributed processing state across all machines in the application. Admin servers are provided as part of the Tuxedo distribution.
application (DACS, Tuxedo)	System of cooperating processes configured for a specific function to be run (in a distributed fashion) by Tuxedo. Also used in a more general sense to refer to all objects included in one particular `ubbconfig` file (machines, groups, servers) and associated shared memory resources, qspaces, and clients.
application server	Server that provides functionality to the application.

TABLE IV: TECHNICAL TERMS (CONTINUED)

Term	Description
backup (component)	System component that is provided redundantly. Backups exist on the machine, group, server, and services level. Appropriate backups are configured to seamlessly take over processing as soon as a primary system component fails or becomes unavailable.
boot	Action of starting a server process as a memory-resident task. Booting the whole application is equivalent to booting all specified server processes (admin servers first, application servers second).
client	Software module that gathers and presents data to an application; it generates requests for service and receives replies. This term can also be used to indicate the requesting role that a software module assumes by either a client or server process.[1]
DACS machines	Machines on a Local Area Network (LAN) that are explicitly named in the `*MACHINES` and `*NETWORK` sections of the `ubbconfig` file. Each machine is given a logical reference (see LMID) to associate with its physical name.
data monitors	Class of application servers that monitor data streams and data availability, form data intervals, and initiate a sequence of general processing servers when a sufficiently large amount of unprocessed data are found.
dequeue	Remove a message from a Tuxedo queue.
enqueue	Place a message in a Tuxedo queue.
forwarding agent	Application server *TMQFORWARD* that acts as an intermediary between a message queue on disk and a group of processing servers advertising a service. The forwarding agent uses transactions to manage and control its forwarding function.
generalized processing server	DACS application server (*tuxshell*) that is the interface between the DACS and the Automatic Processing software. It executes application programs as child processes.
instance	Running computer program. An individual program may have multiple instances on one or more host computers.

TABLE IV: TECHNICAL TERMS (CONTINUED)

Term	Description
LMID	Logical machine identifier: the logical reference to a machine used by a Tuxedo application. LMIDs can be descriptive, but they should not be the same as the UNIX hostname of the machine.
Master (machine)	Machine that is designated to be the controller of a DACS (Tuxedo) application. In the IDC application the customary logical machine identifier (LMID) of the Master is THOST.
message interval	Entry in a Tuxedo queue within the qspace referring to rows in the **interval** or **request** tables. The DACS programs ensure that **interval** tables and qspace remain in synchronization at all times.
message queue	Repository for data intervals that cannot be processed immediately. Queues contain references to the data while the data remains on disk.
partitioned	State in which a machine can no longer be accessed from other DACS machines via IPC resources *BRIDGE* and *BBL*.
qspace	Set of message queues grouped under a logical name. The IDC application has a primary and a backup qspace. The primary qspace customarily resides on the machine with logical reference (LMID) QHOST.
server	Software module that accepts requests from clients and other servers and returns replies.[2]
server group	Set of servers that have been assigned a common *GROUPNO* parameter in the `ubbconfig` file. All servers in one server group must run on the same logical machine (LMID). Servers in a group often advertise equivalent or logically related services.
service	Action performed by an application server. The server is said to be advertising that service. A server may advertise several services (multiple personalities), and several servers may advertise the same service (replicated servers).
shutdown	Action of terminating a server process as a memory-resident task. Shutting down the whole application is equivalent to terminating all specified server processes (application servers first, admin servers second) in the reverse order that they were booted.

TABLE IV: TECHNICAL TERMS (CONTINUED)

Term	Description
SRVID	Server identifier: integer between `1` and `29999` uniquely referring to a particular server. The *SRVID* is used in the `ubbconfig` file and with Tuxedo administrative utilities to refer to this server.
transaction	Set of operations that is treated as a unit. If one of the operations fails, the whole transaction is considered failed and the system is "rolled back" to its pre-transaction processing state.
tuxpad	DACS client that provides a graphical user interface for common Tuxedo administrative services.
`ubbconfig` file	Human readable file containing all of the Tuxedo configuration information for a single DACS application.

1. Tuxedo clients send and receive messages to and from a server, queue messages to a Tuxedo queue, or remove messages from a Tuxedo queue.
2. Tuxedo servers are booted and shut down by the DACS and may run on a remote machine. Servers may be supplied by the Tuxedo distribution (upper case names) or by application programmers (lower case names).

Chapter 1: Overview

This chapter provides a general overview of the DACS software and includes the following topics:

- Introduction
- Functionality
- Identification
- Status of Development
- Background and History
- Operating Environment

Chapter 1: Overview

INTRODUCTION

The software of the IDC acquires time-series and radionuclide data from stations of the International Monitoring System (IMS) and other locations. These data are passed through a number of automatic and interactive analysis stages, which culminate in the estimation of location and in the origin time of events (earthquakes, volcanic eruptions, and so on) in the earth, including its oceans and atmosphere. The results of the analysis are distributed to States Parties and other users by various means. Approximately one million lines of developmental software are spread across six CSCIs of the software architecture. One additional CSCI is devoted to run-time data of the software. Figure 1 shows the logical organization of the IDC software. The Distributed Processing CSCI technically includes the DACS. However, in practice, the DACS is synonymous with the Distributed Processing CSCI. The DACS consists of the following CSCs:

- Application Services

  This software consists of the SAIC-supplied server and client processes of the DACS.

- Process Monitoring and Control

  This software consists of scripts and GUIs that control the way the DACS operates.

- Distributed Processing Libraries

  This software consists of libraries common to the DACS processes.

- Distributed Processing Scripts

  This software consists of a few utilities that create and manage certain aspects of the DACS.

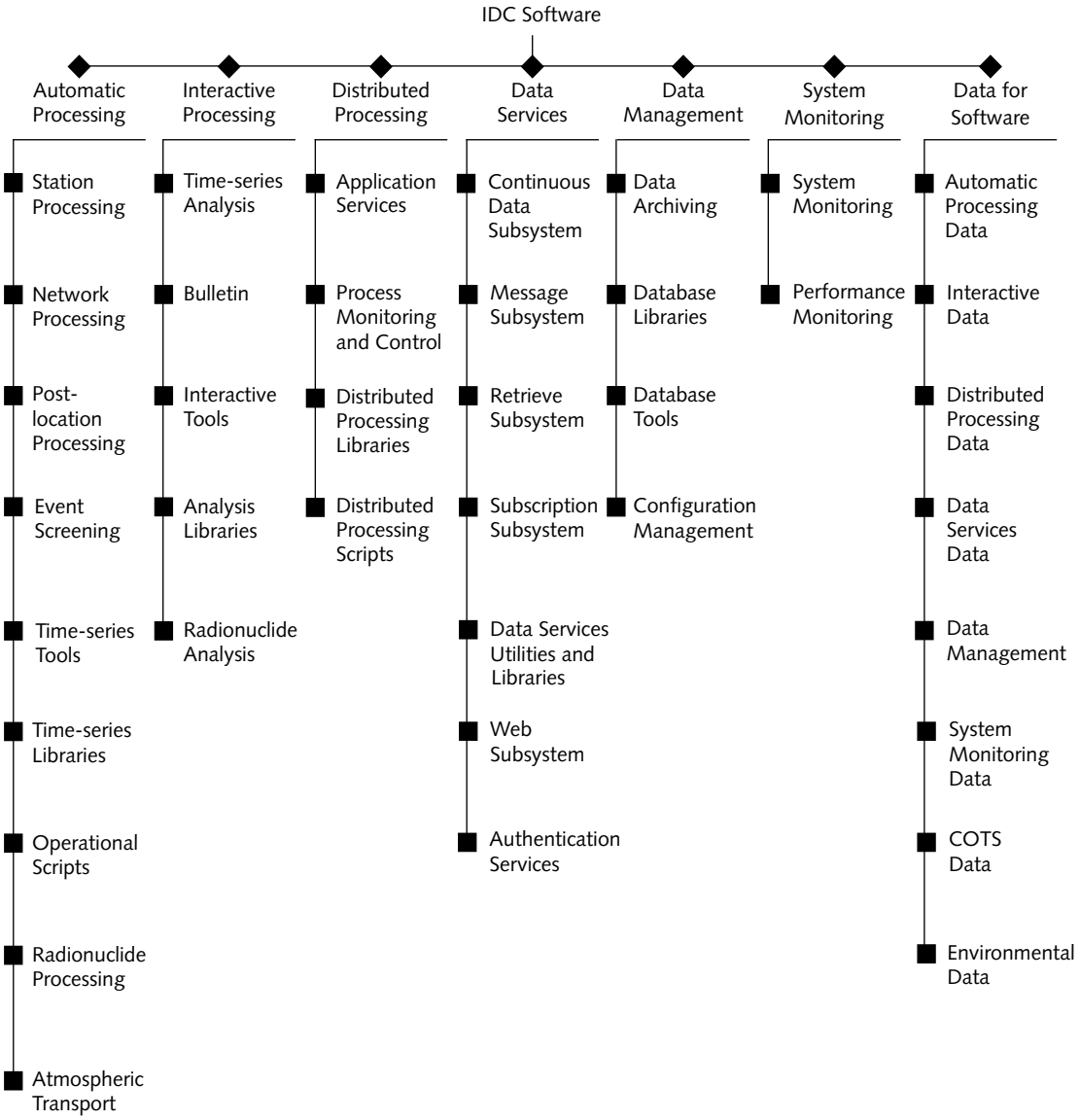


FIGURE 1. IDC SOFTWARE CONFIGURATION HIERARCHY

The DACS is the software between the operating system (OS) and the IDC application software. The purpose of this "middleware" is to distribute the application software over several machines and to control and monitor the execution of the various components of the application software.

Figure 2 shows the relationship of the DACS to other subsystems of the IDC software.

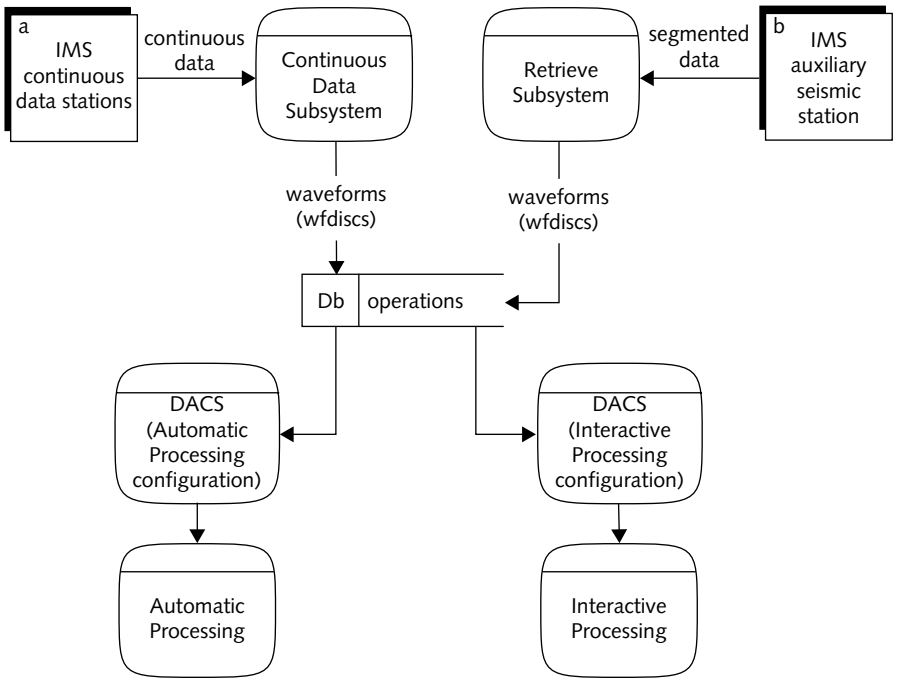


FIGURE 2. RELATIONSHIP OF DACS TO OTHER SUBSYSTEMS OF IDC SOFTWARE

The Continuous Data Subsystem receives data from primary seismic, hydroacoustic, and infrasonic (S/H/I) stations. The Retrieve Subsystem receives data from auxiliary seismic stations. The data consists of ancillary information stored in the ORACLE operations database and binary waveform files stored on the UNIX filesystem. The ancillary information consists of rows in the **wfdisc** table and each row

includes file pointers to raw waveform data. Within the IDC software, the DACS is deployed in two separate application instances. The DACS supports both automatic and interactive processing. The DACS addresses different needs of the software within each of these CSCIs.

Figure 3 shows key features of the DACS that support the Automatic Processing software.

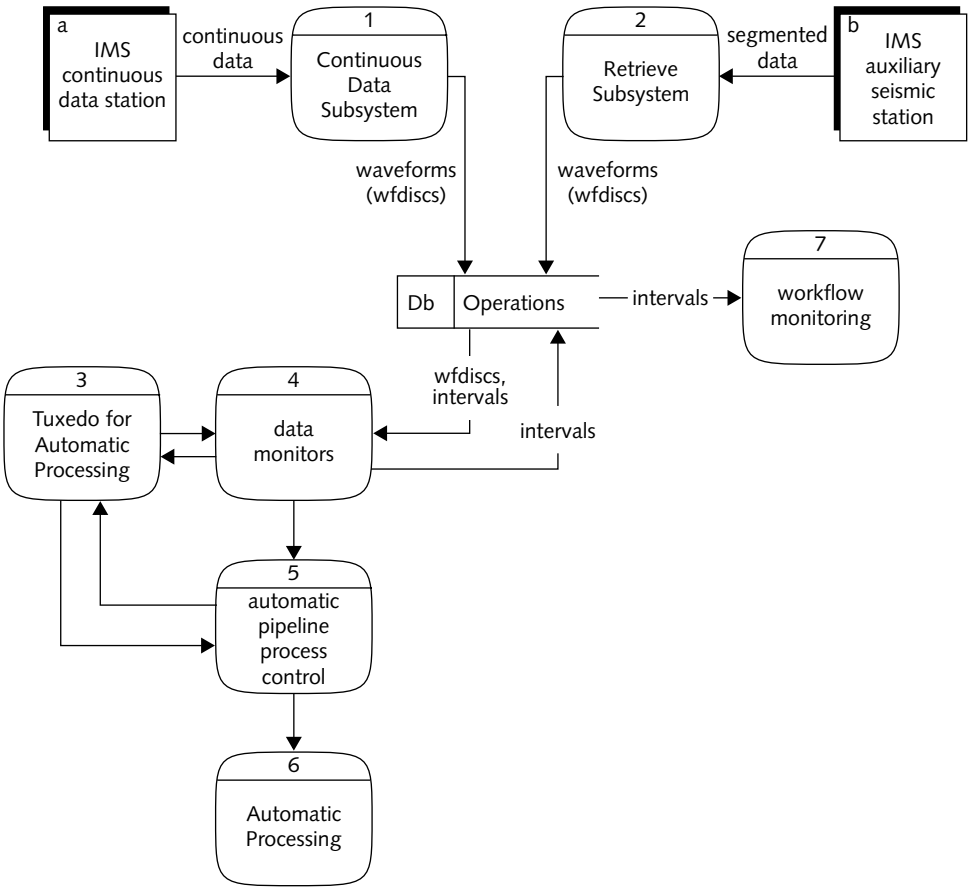


FIGURE 3. DACS APPLICATION FOR AUTOMATIC PROCESSING

In support of Automatic Processing, the DACS is a queue-based system for scheduling a sequence of automated processing tasks. The processing tasks collectively address the mission of the Automatic Processing software, while the DACS adds a non-intrusive control layer. The DACS supports sequential, parallel, and compound sequences of processing tasks, collectively referred to as processing pipelines. These processing pipelines are initiated by the DACS data monitor servers, which query the database looking for newly arrived data. Confirmed data results in new processing intervals that are stored in the database and the DACS queuing system.[1] The database intervals record the state of processing, and this state is visually displayed through the GUI-based *WorkFlow* monitoring application.

Figure 4 shows key features of the DACS application that supports the Interactive Processing software.

In support of Interactive Processing, the DACS is a messaging-based system, which enables data sharing between Interactive Tools. The DACS allows separate programs to exchange messages in near real-time. The DACS provides some management of the Interactive Tools by automatically invoking a requested program when needed. This feature allows an analyst to easily summon the processing resources of occasionally used auxiliary programs. A DACS monitoring utility confirms that processes are running and accepting messages. In support of Interactive Processing, the DACS also supports interactive requests to certain Automatic Processing applications.

1. The DACS queuing system is not shown the figure.

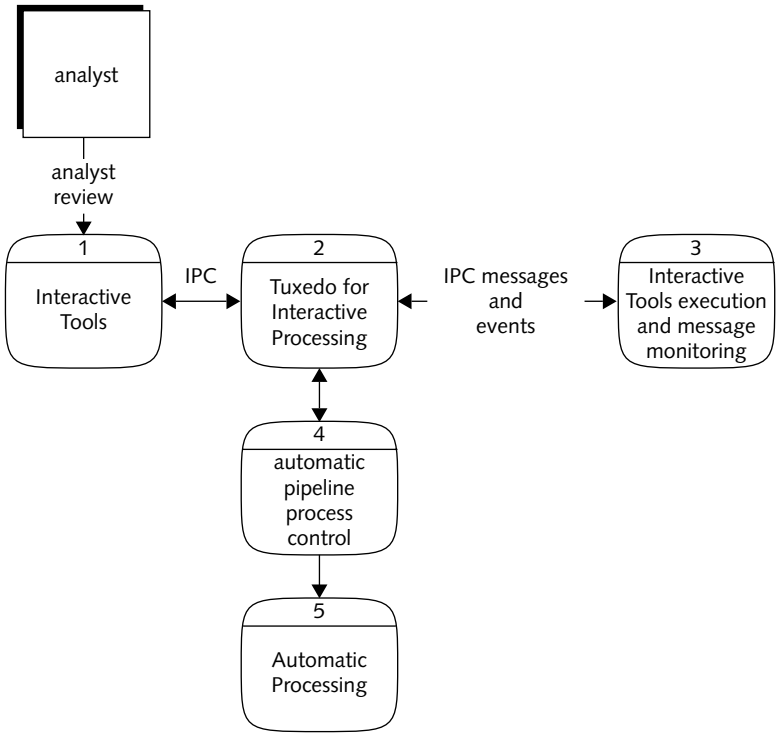


FIGURE 4. DACS APPLICATION FOR INTERACTIVE PROCESSING

FUNCTIONALITY

Figure 5 shows the concept of middleware. The DACS coordinates the execution of various application programs on a network of computers, by controlling these application programs on each machine and using the underlying operating system to maintain contact. The UNIX operating system contains some tools for distributed command execution (the suite of remote commands: *rsh*, *rusers*, *rcp*, *...)* but these lack the extended functionality necessary to support a highly available automatic application. In particular these tools intrinsically do not support process monitoring, process and resources replication and migration, and transactions, which are all important elements in a highly available and fault-tolerant distributed application.

Figure 5 shows how the DACS controls the application software that is running on several machines in a distributed fashion. The individual instances of the DACS coordinate among themselves using features of the underlying operating system and the LAN connecting the machines.

The DACS provides UNIX process management, failure retries, controlled start up and shut down, priority processing, run-time reconfiguration, a monitoring interface, and fault-tolerant operations. All of these functions are supported across a distributed computing platform.

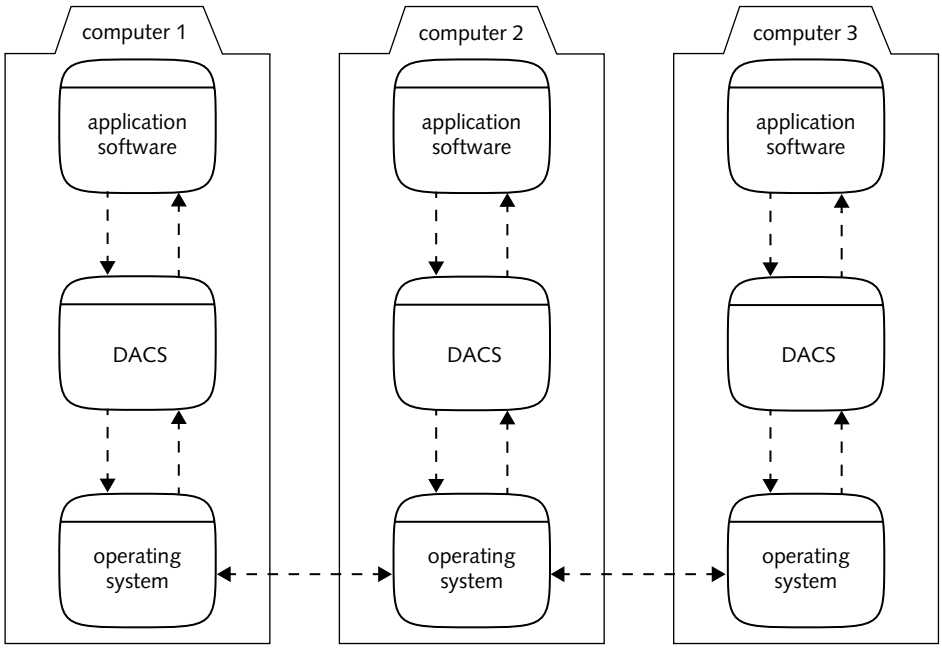


FIGURE 5. DACS AS MIDDLEWARE

The operating system used at the IDC is Solaris, a version of UNIX by Sun Microsystems; the application software is the SAIC-supplied software, and the DACS middleware is a product called Tuxedo, which is provided by BEA. Tuxedo is widely used for banking applications and other branches of industry that maintain distributed applications (for example, phone companies, courier services, and chain

retailers). Tuxedo is a powerful and versatile product of which each application typically uses only a part. This document does not provide an introduction to the full scope of Tuxedo (see [And96] and [BEA96]). Instead, only those features of Tuxedo with a direct bearing on the IDC software are included.

Tuxedo is a transaction manager that coordinates transactions across one or more transactional resource managers. Example transactional resource managers include database servers such as ORACLE and the queueing system that is included with Tuxedo.[2] This queueing system is used extensively by the DACS for reliable message storage and forwarding within the IDC Automatic and Interactive Processing software. The disk-based queues and the database maintain the state of the system during any system or process failure. Tuxedo also provides extensive backup and self-correcting capability, so that network interruptions or scheduled maintenance activity do not disrupt processing.

IDENTIFICATION

The DACS components are identified as follows:

- *birdie*
- *dbserver*
- *dman*
- *interval_router*
- *libipc*
- *msg_window*
- *operate_admin*
- *ProcessInterval*
- *qinfo*

2. The DACS currently does not use Tuxedo for coordinating or managing ORACLE database transactions. The DACS relies upon the native Generic Database Interface (GDI) API (*libgdi*) for all database operations. As such, the DACS coordinates database and Tuxedo queuing transactions within the specific server implementation and without automatic Tuxedo control. Inherent coordination of database and queuing transactions (for example, two phase commits) would require passing ORACLE transactions through Tuxedo.

- *recycler_server*
- *schedclient*
- *schedule_it*
- *scheduler*
- *SendMessage*
- *ticron_server*
- *tin_server*
- *tis_server*
- *tiseg_server*
- *tuxpad*
- *tuxshell*
- *WaveGet_server*
- *WorkFlow*

STATUS OF DEVELOPMENT

This document describes software that is for the most part mature and complete.

BACKGROUND AND HISTORY

A previous implementation of the DACS, based upon the Isis distributed processing system, was deployed into operations at the PIDC at the Center for Monitoring Research (CMR) in Arlington, Virginia, U.S.A. in the early 1990s. The current Tuxedo-based DACS has been used at the PIDC and the International Data Centre of the Comprehensive Nuclear Test Ban Treaty Organization (CTBTO) in Vienna, Austria since the spring of 1998. The graphical operator console, *tuxpad*, was deployed during 1999, and the DACS scheduling system was completely redesigned in early 2000.

OPERATING ENVIRONMENT

The following paragraphs describe the hardware and COTS software required to operate the DACS.

Hardware

The DACS is highly scalable and is designed to run on Sun Microsystems SPARC workstations/SPARC Enterprise servers. The DACS for automatic processing runs on a distributed set of machines that can scale from a handful of machines to tens of machines depending on the data volume and available computing resources. The DACS for interactive processing is most typically run in a stand-alone single SPARC workstation configuration. SPARC workstation and server models are always changing, but a representative workstation is the SPARC Ultra 10, and a representative Enterprise Server is the SPARC Ultra Enterprise 4,000 configured with six Central Processing Units (CPUs). Typically, the hardware is configured with between 64-1,024 MB of memory and a minimum of 10 GB of magnetic disk. The required disk space is defined by other subsystems because the DACS imposes relatively minor disk space requirements with the one exception being in server process logging, which shares significant disk space usage requirements with other CSCIs. The DACS relies upon other system infrastructure and services including the LAN, Network File System (NFS), the ORACLE database server, and the mail server.

Commercial-Off-The-Shelf Software

The software is designed for Solaris 7, ORACLE 8i, and Tuxedo 6.5.

Chapter 2: Architectural Design

This chapter describes the architectural design of the DACS and includes the following topics:

- Conceptual Design
- Design Decisions
- Functional Description
- Interface Design

Chapter 2: Architectural Design

CONCEPTUAL DESIGN

The DACS was designed to address requirements for reliable distributed processing and message passing within the IDC System. The requirements include a number of processing and control features necessary for reliable automatic processing across a distributed network of computers. The message passing requirements for Interactive Processing entail features for passing messages between Interactive Tools and managing the Interactive Tools session.

Figure 6 shows the conceptual data flow of the DACS for Automatic Processing. Tuxedo provides the core distributed processing environment in the DACS. Tuxedo servers are present on all DACS machines. This is shown at the bottom of Figure 6 where Tuxedo queuing, transactions, and process monitoring interact with all of the DACS functions. The DACS monitors the database for data, creates processing intervals (characterized by the start times and end times) subject to data availability (process 2), and manages a pipeline sequence of processing tasks for each interval. The data monitor servers are called on a recurring basis by a scheduling server (process 1), which manages the scheduling and execution of the data monitor services based upon user parameters and input from the data monitors. New processing intervals result in a new pipeline processing sequence that consists of one or more processing tasks. The processing interval information is placed in both the database and Tuxedo queues. Each processing interval contains a *state* field, which is set by the DACS to reflect the current processing state of the interval. System operators can monitor the progress of Automatic Processing by collectively monitoring a time window of intervals in the database. Such process workflow monitoring (process 4) is conveniently presented through a GUI-based display, which renders time interval states as colored bricks.

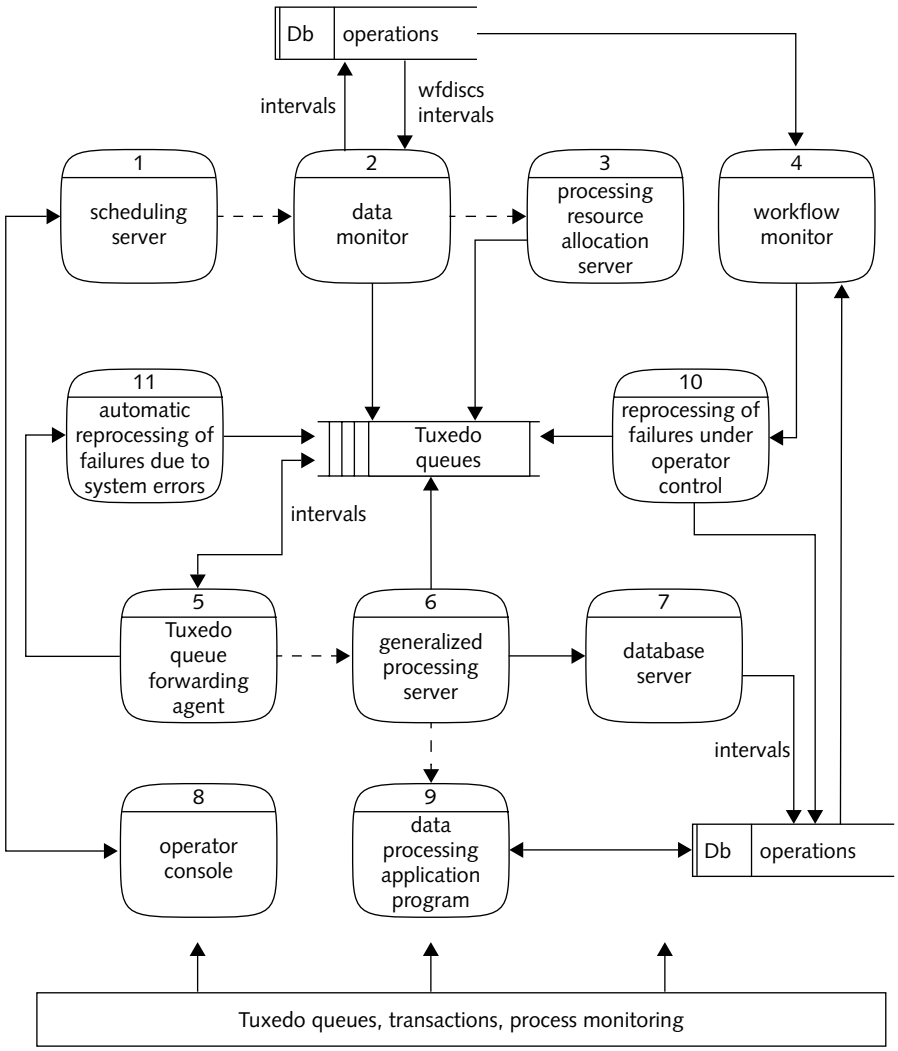


FIGURE 6. CONCEPTUAL DATA FLOW OF THE DACS FOR AUTOMATIC PROCESSING

Interval data are reliably stored in Tuxedo disk queues, which will survive machine failure. The data monitor servers can enqueue the interval data directly into a Tuxedo queue where the queue name is user defined. Optionally, a processing resource allocation server can enqueue interval data into one queue from a set of

possible queues, the selection being a function of the interval type or name (process 3 in Figure 6). A Tuxedo queue forwarding server dequeues the interval data from a Tuxedo queue within a transaction (process 5). The queue forwarder passes the DACS generalized processing server the interval data as part of a service call (process 6). The generalized processing server calls one or more processing applications, which subject the processing interval to the automatic processing task (process 9). The generalized processing server manages the execution of the processing task and handles successful or failed runs and timeouts. Failed processing intervals as well as timeout of the application program result in a transaction rollback of the queue interval by the Tuxedo queue forwarder and a retry, which repeats the queue forwarding sequence (processes 5, 6, 7, and 9). Successful processing intervals result in an enqueue of the updated interval into another downstream Tuxedo queue and a transactional commit of the original queue interval dequeued by the Tuxedo queue forwarder. The downstream Tuxedo queue manages the next step in the pipeline processing sequence, which repeats the queue forwarding sequence (processes 5, 6, 7, and 9). The generalized processing server manages the interval data in the database by updating the interval state to reflect the current processing state. The actual database update is handled by the generalized database application server, which retains one connection to the database while multiplexing database access to a number of generalized processing servers (process 7). Queue intervals that fail due to system errors (for example, machine crash) can be directed to a system-wide error queue from where they are automatically recycled back into service by the automatic reprocessing server (process 11).

The system operator can control DACS via the GUI-based operator console (process 8). Control includes complete DACS bootup or shut down, boot and shut down on a machine, process-group or process-server-basis control of the DACS scheduling system, and monitoring of Tuxedo queues. The system operator can also manually reprocess failed intervals via a feature of the workflow monitoring system (process 10).

Figure 7 shows the conceptual data flow of the DACS for Interactive Processing, using as an example a request for frequency-wavenumber (*Fk*) analysis of a signal. Here, the DACS supports the asynchronous messaging between Interactive Tools, manages the interactive session by monitoring messages and Interactive Tools

within the session, and starts the Interactive Tools on demand. All messages exchanged between the Interactive Tools pass through Tuxedo disk queues. Storing messages within a disk-based Tuxedo queue ensures that the messaging is asynchronous, because the message send and receive are part of separate queuing operations and transactions. Asynchronous messaging allows for one Interactive Tool (process 1) to send a message to another Interactive Tool that is not currently running. A DACS application tracks all message traffic through Tuxedo IPC events, (process 2). This application provides execution on demand for any Interactive Tool that has been sent a message, and is not currently running in the analyst's interactive session (process 3).

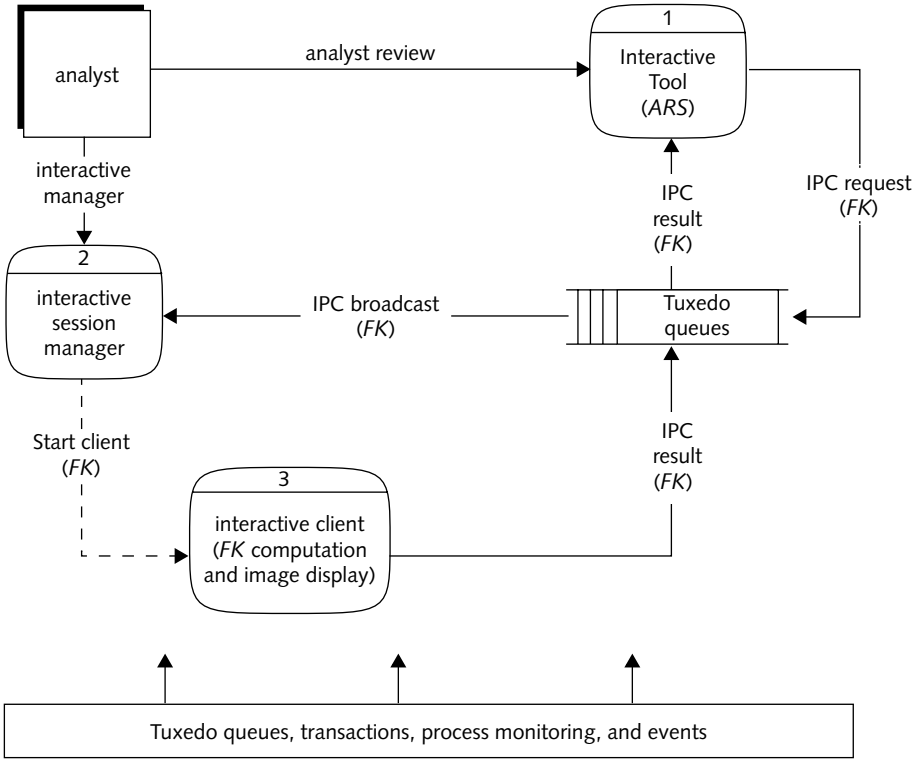


FIGURE 7. CONCEPTUAL DATA FLOW OF DACS FOR INTERACTIVE PROCESSING

DESIGN DECISIONS

All design decisions for the DACS are measured against and can be traced to the significant reliability requirements for Automatic Processing. In general, the DACS must provide fault tolerance and reliability in case of machine, server, and application failures. Fundamentally, all processing managed by the DACS must be under transaction control so that processing tasks can be repeated for a configured number of retries, declared failed following a maximum number of retries, and forwarded for further processing after one and only one successful run.

The decision to introduce a reliable queuing system addresses many of the fault-tolerance requirements because all processing is managed through reliable disk queues under transaction control. The DACS is designed around the Tuxedo distributed processing COTS product to satisfy the requirements to support automatic failover in the case of hardware and software failures.

The decision to use Tuxedo for the message passing requirement for the Interactive Tools was based upon the preference to have a unified distributed processing solution for both Automatic Processing and Interactive Processing. In addition, the Interactive Tools rely upon some limited access to Automatic Processing for on-the-fly signal processing. Such a requirement further justifies a single unified distributed processing solution. However, a Tuxedo implementation for Interactive Processing could be considered an overly heavy-weight solution because the features of the COTS product far surpass the fairly limited message passing and interactive session management requirements.

Programming Language

Each software unit of the DACS is written in the C programming language unless otherwise noted in this document. The *tuxpad* script is implemented using the Perl scripting language.

Global Libraries

The software of the DACS is linked to the following shared development libraries: *libaesir*, *libgdi*, *libipc*, *libpar*, *libstdtime*, and *libtable*.

The software of the DACS is linked to a number of standard system libraries, the bulk of which are required for X11 Window GUI-based applications, such as *WorkFlow*.

The software is also linked to several Oracle COTS libraries indirectly through run-time linking by *libgdi*. The software is linked to the following Tuxedo COTS libraries: *libbuft*, *libfml*, *libfml32*, *libgp*, *libtux*, and *libtux2*.

Database

See "Database Schema Overview" on page 27 for a description of database tables and usage by DACS.

Interprocess Communication (IPC)

By its very nature of being a distributed processing system, the DACS uses and implements various types of IPC and IPC resources. All Tuxedo queuing operations are a form of IPC message passing across machines. Tuxedo provides the *BRIDGE*[1] server, which runs on each distributed machine in the DACS and provides a single point for all Tuxedo-based distributed message sends and message receives. The *libipc* messaging library implements a message passing API based upon Tuxedo queuing. The Tuxedo system makes extensive use of the UNIX system IPC resources including: shared memory, message queues (memory-based), and semaphores. Finally, the DACS relies upon the ORACLE database for another type of IPC via creation/update, and read, to the **interval**, **request**, **timestamp**, and **lddate** tables.

1. The *BRIDGE* server is not included or required for stand-alone Tuxedo applications because all messaging is local to one machine. The current configuration of the DACS for Interactive Processing is standalone, and as such, the *BRIDGE* server is not part of the application.

Filesystem

The DACS uses the UNIX filesystem for reading user parameter files, writing log files, hosting the Tuxedo qspaces and queues as well as the Tuxedo transaction log files. The list of (*libpar*-based) parameter files is extensive and in general each DACS server or client reads one or more parameter files. The DACS servers are routinely deployed in various instances that necessitate distinct parameter files based upon the program's canonical parameter files.

The DACS writes log files at both the system and application level. System-level log files are written by Tuxedo and one such User Log (ULOG) file exists per machine. System-level errors and messages are recorded in these files. The individual ULOGS are copied to a central location (CLOGS) by application-level scripts. Application-level log files are written by DACS servers and clients to record the progress of processing.

Several special system-wide files are required for the DACS. These files include Tuxedo transaction log files (tlogs), qspace files, and the Tuxedo system configuration file (`ubbconfig`), which defines the entire distributed application at the machine, group, server, and service level.

UNIX Mail

The DACS relies upon mail services for automatic email message delivery to system operators when the pending messages overflow in Tuxedo queues.

FTP

The DACS does not directly use or rely upon File Transfer Protocol (FTP).

Web

A Web- and Java-based Tuxedo administration tool is available for administration of the DACS. However, this tool is not used because the custom DACS operator console, *tuxpad*, is preferred over the Tuxedo Web-based solution.

Design Model

The design of the DACS is primarily determined by the fault tolerance and reliability requirements previously described. This section presents a detailed description of some of the key design elements related to the DACS servers and services, namely reliable queuing, transactions, fault-tolerant processing via backup servers, and queue-based pipeline processing for Automatic Processing.

Figure 8 shows the logical relations between message queue, service, server, and host. The message queue (A) contains a number of requests for service *A* (for example, data intervals to be processed by the application program *DFX*). On three different hosts (physical UNIX machines host 1, host 2, and host 3), three servers (A1, A2, and A3) are running, each of which is capable of providing the service *A*. The DACS assures that each service request goes to one and only one server, and is eventually removed from the message queue only after processing is complete.

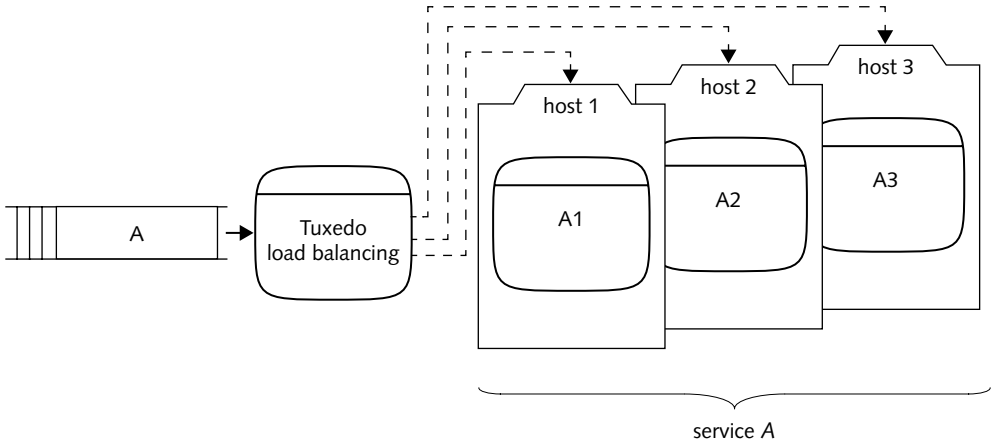


FIGURE 8. PROCESSING REQUESTS FROM MESSAGE QUEUE

Figure 9 shows a transaction as one step in a series of processing steps to be applied to data intervals. It shows a processing server A*n* between a message queue A (its source queue) and a message queue B (its destination queue). The processing server advertises service *A* and is capable of spawning a child process *a*, the automated processing program that actually provides the service.

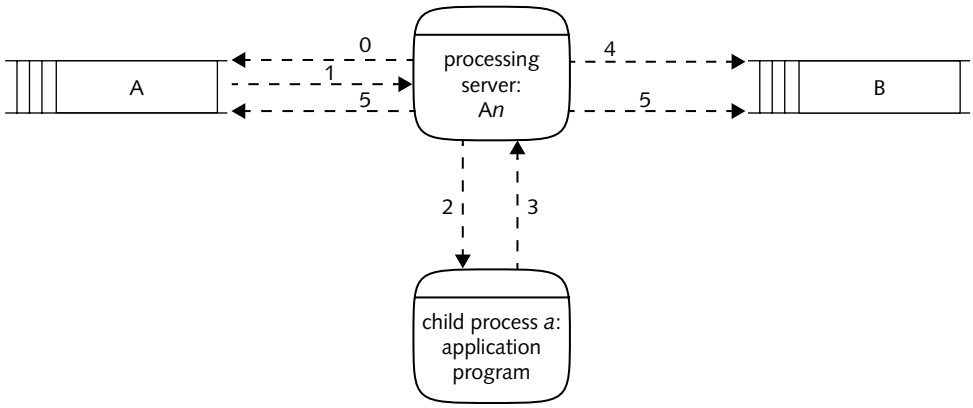


FIGURE 9. TRANSACTION IN DETAIL

Assuming that queue A contains at least one message, the first step of the transaction (step 0) is to provisionally remove the uppermost message from queue A. In step 1, information is extracted from the message and sent to processing server A*n*. Server A*n* spawns a child process *a* and passes some of the information previously extracted from the message to the child process (step 2). The information passed to the child process typically designates a data interval on which the service *a* is to be performed. The child process processes the data and signals its completion to the processing server (step 3). If the data were processed successfully, a message is placed provisionally in queue B (step 4). The concluding step 5 commits (finalizes) the changes to the source queue A and the destination queue B.

If a failure occurs on any of the steps (0 through 5), the entire transaction is "rolled back," which means that the provisional queueing operations in step 0 and step 4 and any other change in the state of the system (for example, in the database) are

reversed. The rollback applies not only to failures of the actual processing by the child process, but also to the queueing operations, the actions of the processing server, and to the final commit.

Figure 10 provides further detail on the interface between the message queue and the processing server. It shows that a forwarding agent mediates between the two. Only the forwarding agent (a Tuxedo-supplied server called *TMQFORWARD* described in "Application Servers" on page 43) handles the queue operations. Figure 10 omits the transactional components of the operation for simplicity. A "Reply Queue" feature is provided by Tuxedo but is not exploited for building pipelines in the IDC application; instead, the processing server places messages directly in the next queue of the processing sequence (queue B in Figure 9, not shown on Figure 10).

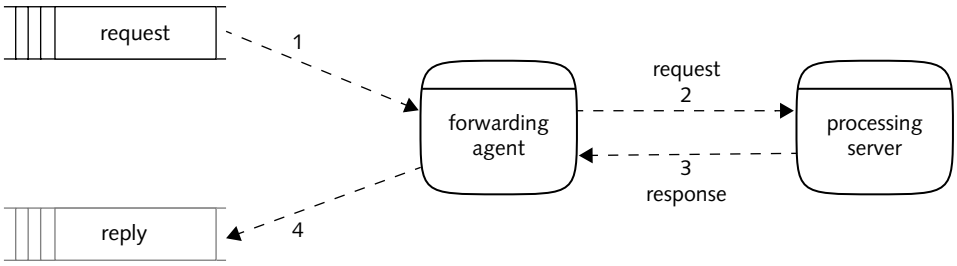


FIGURE 10. FORWARDING AGENT

Distribution and Backup Concept

Even with multiprocessor machines, no single computer within the IDC has the capacity to run the entire IDC software. Therefore, the application must use several physical machines. Moreover, the number of data sources exceeds the number of available processors by an order of magnitude, and processing the data from a single source requires substantial computing resources. This combination suggests a queueing system to distribute the processing load over both space and time.

The constraints imposed by the computer resources lead to the design of the IDC software as a distributed application with message queues. Processing is divided into a number of elementary services. These services are provided by server programs, which run on a number of machines under the control of the DACS. Message queues are interspersed between the elementary services.

The distribution scheme is based on the following objectives:

- Capacity Mapping

  All machines should be loaded in accordance with their capacities.

- Load Limitation

  No component of the system should be allowed to overload to a point where throughput would suffer.

- Load Balancing

  All machines should be used to approximately the same level of their total capacity.

- Minimization of Network Traffic

  Whenever possible, mass data flow over the LAN should be avoided. For example, detection processing should usually occur on the machine that holds the data in a disk loop.

- Catchup Capability

  Some extra capacity (in terms of processing speed, *n* times real time) should be reserved for occasions when processing must "catch up" with real time.

- Single-Point-of-Failure Tolerance

  The system should withstand any single failure (hardware or software) and allow scheduled maintenance of individual (hardware or software) components without interrupting processing, or, if interruption is inevitable, with a seamless resumption of processing.

These objectives cannot always be met. Trade-offs between objectives arise given the fact that hardware and development resources are finite.

Pipelines

During automatic processing, the same data interval is processed by a number of application programs in a well-defined processing sequence known as a "pipeline." For example, station processing consists of the application programs *DFX* and *StaPro*, and network processing for SEL1 is comprised of *GA_DBI*, *GAassoc*, *GAconflict*, and *WaveExpert*.

Figure 11 shows how a pipeline can be constructed. The data monitor checks the state of the database and creates intervals and enqueues messages when a sufficient amount of unprocessed data are present or when some other criterion is fulfilled (for example, a certain time has elapsed). Each processing server receives messages from its source queue and spawns child processes that perform the actual processing step in interaction with the database. After completion, the processing server places a new message in its destination queue, which in turn is the source queue for the next processing server downstream and so on, until messages finally arrive in the "done" queue.

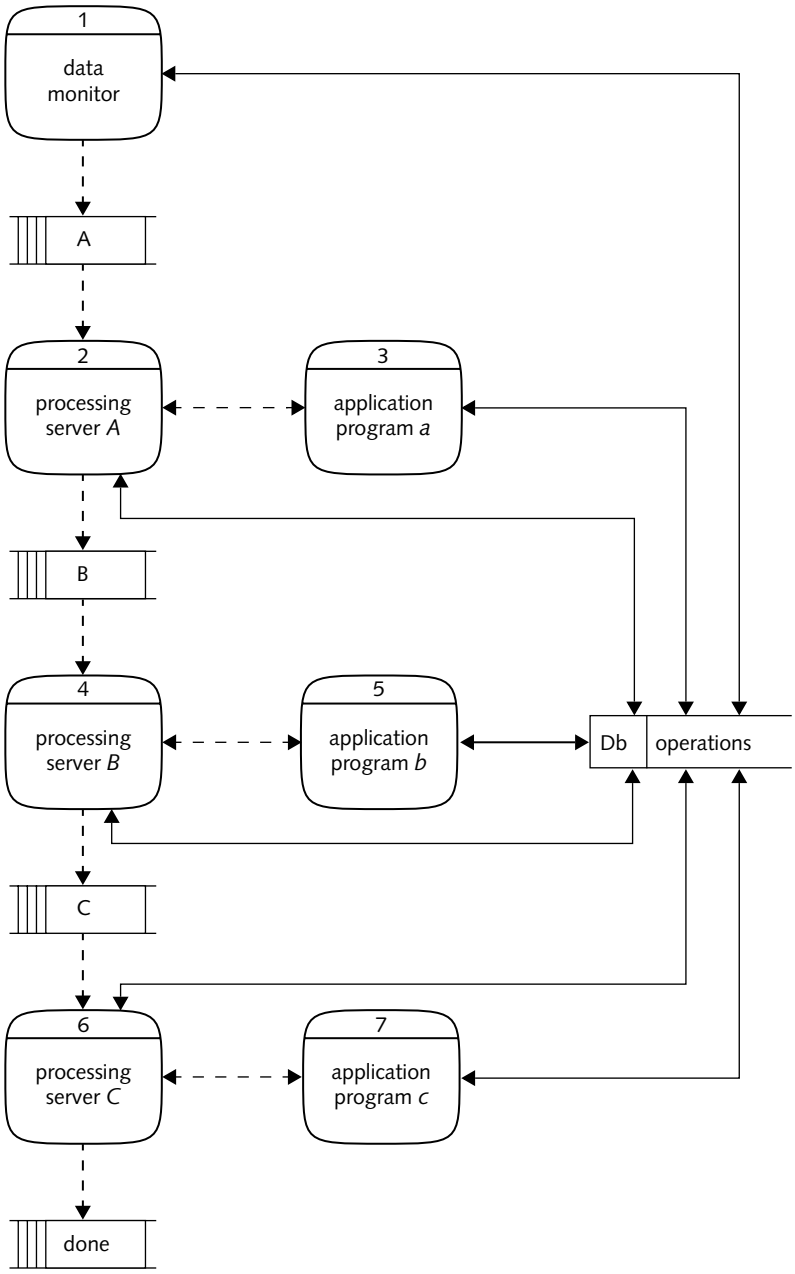


FIGURE 11. CONSTRUCTION OF A PIPELINE

Database Schema Overview

The DACS uses the ORACLE database for the following purposes:

- To obtain data availability (acquired waveform data, submitted data requests)
- To obtain interval processing progress via queries to the **interval** table
- To create processing intervals and requests and update their states
- To obtain and store the DACS processing progress by time (for example, *tis_server* progress)
- To obtain and store specific station **wfdisc**.*endtime* information in an efficient manner
- To obtain network, station, and site affiliation information
- To store and manage unique interval identifier information

Table 1 shows the tables used by the DACS along with a description of their use. The Name field identifies the database table. The Mode field is "R" if the DACS reads from the table and "W" if the system writes/updates to the table.

TABLE 1: DATABASE TABLES USED BY DACS

Name	Mode	Description
affiliation	R	This table is a general mapping table, which affiliates information. The DACS uses the affiliation information to obtain mappings between network and stations and stations and sites during station-based interval creation.
interval	R/W	This table contains the state of all processing intervals that are created, updated, displayed, and managed by the DACS.
lastid	R/W	This table contains identifier values, which the DACS uses to ensure unique **interval**.*intvlid* for each interval created.

TABLE 1: DATABASE TABLES USED BY DACS (CONTINUED)

Name	Mode	Description
request	R/W	This table contains the state of auxiliary waveform requests, which the DACS uses to manage and initiate auxiliary waveform acquisition processing. Optionally, this table is used to create auxiliary station pipeline processing intervals.[1]
timestamp	R/W	This table contains time markers, which the DACS uses to track interval creation progress and to retrieve **wfdisc**.*endtime* by station.
wfdisc	R	This table contains references to all acquired waveform data, which the DACS reads to determine data availability for the creation of processing intervals.

1. The IDC does not currently use this feature.

FUNCTIONAL DESCRIPTION

This section describes the main functions of the DACS. Figure 12, and Figure 13 on page 34, are referenced in the Functional Description.

Distributed Process Monitoring, Reliable Queueing, and Transactions

Tuxedo provides the core distributed processing environment in the DACS. Tuxedo servers are present on all DACS machines. This is shown at the bottom of Figure 12 where Tuxedo queuing, transactions, and process monitoring interact with all of the DACS functions.

The queueing function, transactions, replicated or backup servers, and pipeline processing are described in the previous section. The Tuxedo-supplied distributed process monitoring function involves the real-time monitoring of every DACS server (IDC or COTS supplied) such that the servers are automatically rebooted upon any application failure or crash.

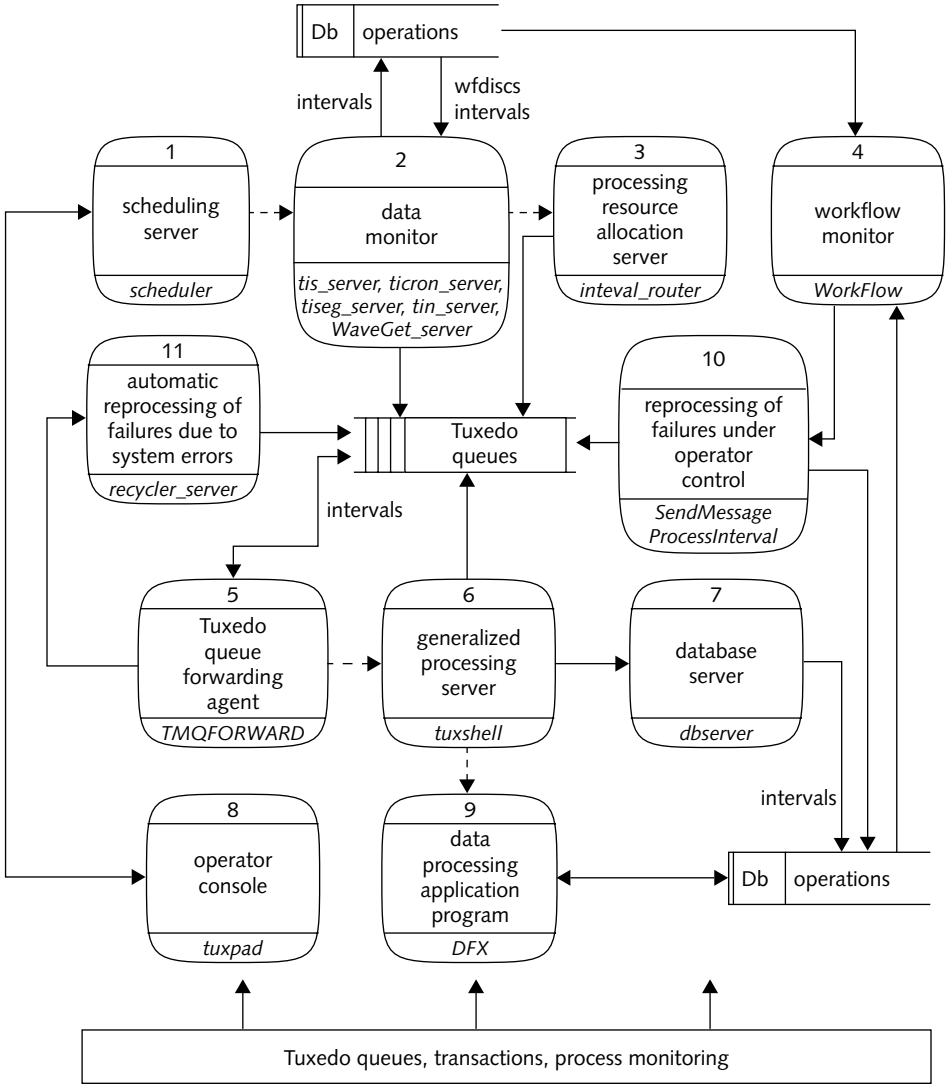


FIGURE 12. DATA FLOW OF THE DACS FOR AUTOMATIC PROCESSING

Data Monitoring

The data monitoring function determines whether new data have become available or if a data condition or state is met. If the monitored condition is met, interval data are inserted into the database, or existing rows are updated from **interval**.*state* to **interval**.*queued*, and the interval information is inserted into Tuxedo queues (process 2 in Figure 12 on page 29). The data monitored in the database varies, and several data monitor servers process the different types of data.

The component *tis_server* monitors S/H/I data delivered from stations that have a continuous, real-time data feed. *tiseg_server* monitors auxiliary seismic station data. *ticron_server* monitors a timestamp value in the database, which tracks the last time the server created a network processing interval. The server forms network processing intervals by time, and so its primary function ensures the timely creation of the network processing intervals. *tin_server* monitors station processing progress by querying the state of a group of stations. The server creates intervals based upon a trade-off between data availability and elapsed time. *WaveGet_server* is a data monitor server that polls the **request** table for auxiliary-station-waveform requests and initiates actions to acquire the requested waveforms.

For each interval created or updated, a data monitor also sends a processing request message to *interval_router* (process 3 in Figure 12 on page 29), or, depending on configuration, bypasses *interval_router* and enqueues the message(s) directly in Tuxedo queues. The Tuxedo queue messages seed the DACS with time-interval-based pipeline processing requests, which are managed by the DACS.

System Scheduling

The system scheduling function provides a centralized server, *scheduler* for automatic data monitor calls and a tool for centralized management of the scheduling system (process 1 in Figure 12 on page 29). The DACS data monitor application servers (for example, *tis_server*, *WaveGet_server*) await service calls from *scheduler*

to perform or complete their data monitoring function and return acknowledgments to *scheduler* following completion of their service cycle. The scheduling system can be controlled by the user via the *schedclient* application.

The *tuxpad* GUI operator console provides a convenient interface to *schedclient*.

Pipeline Processing

The pipeline processing function provides for reliable process sequencing (process 6 in Figure 12 on page 29) and is implemented by the generalized processing server *tuxshell*. Pipeline process sequencing includes application software execution and management within a transactional context. *tuxshell* receives interval messages within a *TMQFORWARD* transaction (process 5 in Figure 12 on page 29). *tuxshell* extracts parameters from the interval message, constructs an application processing command line and then executes and manages the processing application (process 9 in Figure 12 on page 29). The processing application is typically an Automatic Processing program (for example, *DFX*). Processing failures result in transaction rollback and subsequent retries up to a configured maximum number of attempts. Successful processing results in forwarding the interval information via an enqueue into a downstream queue in the pipeline sequence. The state of each interval processed is updated through server calls to the database application server, *dbserver* (process 7 in Figure 12 on page 29).

Workflow Monitoring

The workflow monitoring function provides a graphical representation of interval information in the system database, in particular in the **interval** and **request** database tables (process 4 in Figure 12 on page 29). The monitoring function is implemented by the *WorkFlow* program, which provides a GUI-based operator console for the purpose of monitoring the progress of all automatic processing pipelines in real or near real time. The current state of all processing pipelines is recorded in the *state* column of each row in the **interval** and **request** database tables. Workflow monitoring is primarily a read-only operation. However, failed intervals can be reprocessed under operator control (process 10 in Figure 12 on page 29). The interval reprocessing function is implemented by the *SendMessage* client and *Pro-*

cessInterval script, which collectively change the state in the database of the interval being reprocessed and requeue the interval message to the source queue. These operations manually initiate automatic processing on the interval.

Automatic Processing Utilities

Elements of scalability and reliability in the DACS are provided by several Automatic Processing utilities. Two of these utilities have been described above: *dbserver* updates the database for all **interval.***state* or **request**.*state* updates within the DACS (process 7 in Figure 12 on page 29), and *interval_router* (process 3 in Figure 12 on page 29) routes interval messages created by the data monitor servers to a set of queues as a function of the interval name. System errors such as a machine crash or network failure can and do result in messages that cannot be reliably delivered within the distributed processing system. The DACS message passing is based on Tuxedo disk queues, which safeguard against the loss of messages during system failures.[2] Queue operations that cannot be successfully completed typically result in message redirection to an error queue. These messages are then automatically requeued for reprocessing attempts by *recycler_server* (process 11 in Figure 12 on page 29).

Operator Console

The operator console function provides an interface for controlling the DACS (process 8 in Figure 12 on page 29). This function is implemented by *tuxpad*, a convenient centralized operator console that can be used to control all aspects of the running distributed application.

Interactive Processing

The DACS provides several key functions for Interactive Processing including asynchronous message passing, session management for Interactive Tools, and access to Automatic Processing applications. The Interactive Tools are used by an analyst

2. Tuxedo queue message loss or queue corruption could occur if the physical disk drive hosting the qspace failed.

(see Figure 13) within an interactive session that is typically hosted by a single workstation. Tuxedo is thus configured to run stand-alone on the single workstation, which results in all the DACS processes, queuing, and Automatic Processing being isolated on this machine.[3] The stand-alone machine is still connected to the operational LAN with full access to the database server, and so on. The analyst is principally interested in the review of events formed by Automatic Processing and relies upon the key interactive event review application (process 1), which is implemented by the *ARS* program. In addition, interactive review relies on a collection of Interactive Tools that exchange messages. The DACS supports asynchronous message passing via the *libipc* message passing library. The library is based upon Tuxedo disk queuing, and as such, all messages among the Interactive Tools pass through Tuxedo queues. The DACS also supports management of the interactive session including the ability to start up and shut down Interactive Tools on demand. Interactive session management is implemented by the *dman* client (process 2). For example, a message sent from *ARS* to *XfkDisplay*, via *libipc*, results in both an enqueue of the message to the *XfkDisplay* queue and an IPC event, which is sent to *dman* by *libipc* to broadcast the request for *XfkDisplay* service. *dman* will automatically start the *XfkDisplay* application (process 3) if it is not already running. *dman* monitors all messaging among the Interactive Tools as well as the health of all Interactive Tools within the session. Interactive tools can be manually started or terminated via the *dman* GUI interface. Access to Automatic Processing is provided to a limited degree: Interactive Tools can send messages requesting certain Automatic Processing services for interactive recall processing. This linkage is not shown in Figure 13, but this function was described above as the generalized processing server, *tuxshell* (processes 6 and 9 in Figure 12 on page 29).

3. The stand-alone configuration is a system configuration decision based largely upon the notion of one analyst, one machine. DACS for Interactive Processing could be distributed over a set of workstations through configuration changes.

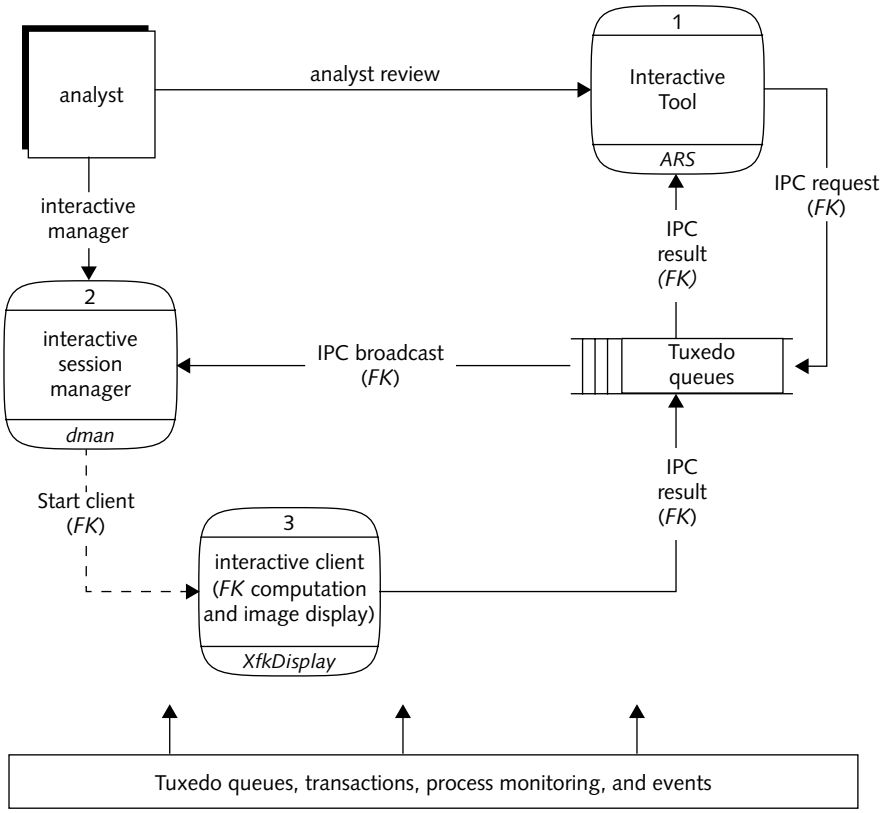


FIGURE 13. DATA FLOW OF THE DACS FOR INTERACTIVE PROCESSING

INTERFACE DESIGN

This section describes the DACS interface with other IDC systems, external users, and operators.

Interface with Other IDC Systems

The DACS controls Automatic Processing by initiating and managing pipeline processing sequences. The DACS relies upon the Continuous Data Subsystem to acquire new sensor data so that new processing time intervals can be generated.

The database serves as the data exchange broker for the DACS and the various Data Services subsystems. The DACS provides message passing and session management to the Interactive Tools within the Interactive Processing System.

Interface with External Users

The DACS has no interface with external users.

Interface with Operators

System operators control and monitor the DACS through *tuxpad* and *WorkFlow* as described above. The DACS for Automatic Processing and Interactive Processing is designed to run unattended and to survive many failure conditions. Ideally, operator control is limited to planned system start up, shut down, and maintenance.

The DACS servers record processing progress such as interval creation and pipeline processing executions on the system-wide logging directory tree. Automatic Processing progress and problem detection and resolution can be ascertained through the inspection and analysis of one or more of the DACS log files. Operators will often be the first to examine the log file; however, developers of the Automatic Processing programs may examine the files in the course of debugging at system level.

Chapter 3: Tuxedo Components and Concepts

This chapter describes the Tuxedo COTS software product including the components and function of Tuxedo used by the DACS and includes the following topics:

- Processing Units
- Tuxedo Components of DACS

Chapter 3: Tuxedo Components and Concepts

PROCESSING UNITS

The DACS consists of the COTS software product Tuxedo and SAIC-developed components. This chapter describes the building blocks of Tuxedo used by the DACS. Table 2 maps the Tuxedo components described in this chapter to the SAIC-developed components. The mapping implies either direct or indirect interaction between the components. The type of interaction is specified by a set of symbols that are defined in the table.

TUXEDO COMPONENTS OF DACS

Listener Daemons (tlisten, tagent)

Listener daemons are processes that run in the background on each DACS machine. Listener daemons are started before and independently of the rest of the distributed application to support the initial application boot on each machine (the bootstrapping of the application).

If an application is distributed, like the DACS for automatic processing, a Tuxedo daemon, *tlisten,* maintains the network connections among the various machines that are part of the application by listening on a particular port. One and only one *tlisten* process must be running on each machine in a distributed application at all times. Without *tlisten,* a machine is not accessible for requests to boot servers.

TABLE 2: MAP OF TUXEDO COMPONENTS TO SAIC DACS COMPONENTS[1]

Tuxedo Component	Data Monitor[2]	scheduler	schedclient	tuxshell	dbserver, interval_ router	recycler _server	WorkFlow, SendMessage[3]	libipc, dman, birdie	tuxpad[4]
tlisten/ tagent[5]	Bs	Bs		Bs	Bs	Bs			
BRIDGE[6]	Sn/Rn	Sn/Rn	Sn/Rn	Sn/Rn	Sn/Rn	Sn/Rn	Sn	Sn/Rn	
BBL/DBBL	Ms	Ms	Mc	Ms	Ms	Ms	Mc	Mc	
TMS/ TMS_QM[7]	Mt	Mt	Mt	Mt	Mt[8]	Mt	Mt	Mt	
TMQUEUE	Eq	Eq/Dq	Eq/Dq	Eq	Eq[9]	Eq	Eq	Eq/Dq	
TMQFOR-WARD		Fs		Fs		Fs			
TMUSREVT				Es				Es/Er	
IPC resources	I	I	I	I	I	I	I	I	
`ubbcon-fig/tux-config`[10]	Ds	Ds		Ds	Ds	Ds			
user logs	Ls	Ls	Ls	Ls	Ls	Ls	Ls	Ls	
transaction logs	Lt	Lt	Lt	Lt	Lt	Lt	Lt	Lt	
queue space	Sq	Sq	Sq	Sq	Sq	Sq	Sq	Sq	

TABLE 2: MAP OF TUXEDO COMPONENTS TO SAIC DACS COMPONENTS[1] (CONTINUED)

Tuxedo Component	Data Monitor[2]	scheduler	schedclient	tuxshell	dbserver, interval_ router	recycler _server	WorkFlow, SendMessage[3]	libipc, dman, birdie	tuxpad[4]
queues	Sm	Sm	Sm	Sm	Sm	Sm	Sm	Sm	
tmloadcf									
tmunloadcf									Gc
tmadmin	Aa	Aa	Aa	Aa	Aa	Aa	Aa	Aa	Aa
qmadmin									Aq

1. Interaction Symbol Definitions:
   Bs (Boots the server)
   Sn/Rn (Sends message over network for server/Receives message via network for server)
   Ms/Mc (Monitors the server with process management/Monitors the client with no process management)
   Mt (Manages servers and clients queue transactions)
   Eq/Dq (Enqueues message for server or client/dequeues message for server or client)
   Fs (Fowards queue-based service call within a queue-based transaction)
   Es/Er (Sends event message for client or server/Receives event message for client or server)
   I (Sends, receives, and stores local messages and state for server and client using IPC resources)
   Ds (Defines server to the application in the `ubbconfig`/`tuxconfig` files)
   Ls (Logs system-level server or client messages to disk)
   Lt (Logs server and client transactions to disk)
   Sq (Stores servers' and clients' queues)
   Sm (Stores server and client queue messages)
   Gc (Generates text version of system configuration that can be parsed for current state of servers, machines, and so on)
   Aa (Administers the application including starting, stopping, and monitoring servers and machines)
   Aq (Administers Tuxedo queuing)
2. Data Monitors include five servers: *tis_server*, *tiseg_server*, *ticron_server*, *tin_server*, and *WaveGet_server*.
3. Only *SendMessage* interacts directly with Tuxedo; *WorkFlow* is strictly a database application.

4. *tuxpad* includes the five scripts: *tuxpad*, *operate_admin*, *schedule_it*, *qinfo*, and *msg_window*. Only *qinfo* uses *qmadmin*.
5. SAIC-supplied DACS servers are started by *tlisten (*via *tagent)* under Tuxedo operator control or under automatic Tuxedo control.
6. All servers and clients (SAIC or Tuxedo supplied) rely upon *BRIDGE* services for inter-machine communication. *tuxpad* scripts execute Tuxedo-supplied and DACS-supplied utilities and clients, but *tuxpad* scripts are not directly connected to the running Tuxedo application.
7. Interaction with the Tuxedo transaction managers is indirect and is handled by Tuxedo on behalf of SAIC DACS components.
8. Queuing transaction is applicable only to *interval_router*.
9. Enqueue operation is applicable only to *interval_router*.
10. The `ubbconfig/tuxconfig` defines IDC servers that are run and managed by the Tuxedo application. IDC clients are not defined in application configuration.

The *tlisten* process is the parent to all Tuxedo servers; its child processes inherit its user identifier (UID), group identifier (GID), and environment. This feature allows the DACS to run under a distinct UID and environment on each machine, provided *tlisten* is started by the user with this UID, in this environment, and the distinct UIDs have been specified in the `*MACHINES` section of the `ubbconfig` file.

To launch other servers, *tlisten* uses *tagent*, which is supplied by Tuxedo. In contrast to *tlisten*, *tagent* is only launched on demand and promptly exits after completing its task.

Administrative Servers

Administrative servers are Tuxedo-supplied servers, which implement the fundamental elements and infrastructure of the distributed application. These include network-based message passing and management of the state of the distributed application, distributed transaction management, and queuing services.

BSBRIDGE and BRIDGE

The bootstrap bridge *BSBRIDGE* is launched by *tlisten* when the user boots the administrative servers on a machine. *BSBRIDGE* prepares the launch of the permanent *BRIDGE* and exits as soon as *BRIDGE* has been established.

BRIDGE manages the exchange of all information between machines (such as the passing of messages). *BRIDGE* remains in the process table until the application is shut down (completely or on the particular machine). If *BRIDGE* crashes or is terminated accidentally, the machine is partitioned (can no longer be accessed from other DACS machines via IPC resources, *BRIDGE*, and *BBL*) and operator intervention is required to restore processing on the machine.

BBL/DBBL

The Bulletin Board Liaison (*BBL*) generates and manages the "Bulletin Board." The Bulletin Board is a section of shared memory in which Tuxedo stores the current state of the application. One copy of the Bulletin Board is on each machine. *BBL* is

launched on each machine after the *BRIDGE* has been established. It remains in the process table until the application is shut down (completely or on the particular machine).

DBBL generates and manages the "Distinguished Bulletin Board," which exists only on the Master machine. *DBBL* is launched on the Master machine at boot and remains in the process table until the application is shut down. The *DBBL* keeps all *BBL*s synchronized so that all machines are in a consistent state across the distributed system. The *DBBL* automatically restarts any *BBL* in the case of a crash or accidental kill. The *BBL* on the Master machine automatically restarts the *DBBL* upon any failure or crash of the *DBBL*. When the Master machine is properly migrated to the backup Master machine, the *DBBL* is also migrated to the new Master machine.

Application Servers

Application servers are Tuxedo-supplied servers, which include application-level infrastructure and services that are necessary for many distributed processing applications. The Tuxedo-supplied infrastructure and services include distributed transaction management, reliable disk-based queuing services, and event message passing services.

TMS/TMS_QM

These application servers manage transactions including the create, commit, rollback, abort, and timeout transactional commands or elements. For each server group the system automatically boots two *TMS*s (Transaction Manager Servers), and for the server groups operating on qspaces the system boots two *TMS_QM*s (*TMS* for Queue Management).

TMQUEUE

TMQUEUE enqueues and dequeues messages from a qspace for other servers (for example, for the data monitors). Each qspace must have at least one instance of *TMQUEUE*. At least one backup instance of *TMQUEUE* per qspace is recommended.

TMQFORWARD

The forwarding agent, *TMQFORWARD,* dequeues messages from a specific disk queue and sends them for processing to a server that advertises the corresponding service. By convention, queue names and service names are identical. In the IDC application the servers advertising processing services are various instances of *tuxshell*, the general application server.[1] *tuxshell* is discussed in "Chapter 4: Detailed Design" on page 47.

Because *TMQFORWARD* works in a transactional mode, it does not commit to dequeueing messages from a queue until the server signals success. Upon any failure, or if a configured time-out value (`-t` on the *TMQFORWARD* command line in the `ubbconfig` file) is reached, *TMQFORWARD* terminates the transaction, requeues the message to the top of the originating queue, and increases the retry count. This recycling action continues until a retry threshold (set at queue creation time) has been exceeded, at which point *TMQFORWARD* drops the message. If all servers advertising the service are busy, *TMQFORWARD* waits for one to become available. If the service is not being advertised, *TMQFORWARD* enqueues the message into the error queue.

TMSYSEVT, TMUSREVT

TMSYSEVT and *TMUSREVT* are servers that act as event brokers. These servers allow communication between application servers and clients and are used only in the interactive DACS application.

1. *TMQFORWARD* can call any server that advertises the same server name as the name of the queue that *TMQFORWARD* monitors. The DACS uses *TMQFORWARD*s that only call *tuxshell* servers.

IPC Resources

Tuxedo uses several IPC resources. These are shared memory, message queues, and semaphores. These resources must be sized correctly within the operating system (in the `/etc/system` file) and are dynamically allocated and freed by Tuxedo at run-time.

Special Files

ubbconfig/tuxconfig

The binary `tuxconfig` file contains the complete configuration of the application in machine-readable form. The Tuxedo operator on the Master machine generates this file by compiling the text file, `ubbconfig`, using the command `tmloadcf`. The Syntax is checked before the compilation. At boot time, the `tuxconfig` binary file is then automatically propagated to all machines in the application. The current state of the configuration of the application can be observed using the command, `tmunloadcf`, or with the *tuxpad* GUI.

User Logs

All Tuxedo processes write routine messages, warnings, and error messages to ASCII user log files `ULOG.`*mmddyy* (with *mmddyy* representing month, day, and year). The log files are kept on a local disk partition for each machine to avoid losing logs or delaying processing due to network problems.

Transaction Logs

Tuxedo tracks all currently open transactions on all machines by recording transaction states in `tlog` files. Consequently, open transactions are not lost, even if a machine crashes. The `tlog` files are binary and have the internal structure of a "Tuxedo device."

Queue Spaces and Queues

The DACS uses the Tuxedo queuing system to store processing requests that have been issued, for example, by a data monitor, but have not yet been executed. These process requests are stored as messages in disk queues. Each queue holds requests for a certain service, for example GAassoc-sel1 or DFX-recall, where the service name matches the queue name. A queue space (or qspace in Tuxedo literature) is a collection of queues. The automated system of the IDC application software works with two qspaces, a primary and a backup, on two different machines, with dozens of queues in each qspace.

Utility Programs

tmloadcf/tmunloadcf

The program *tmloadcf* loads (converts) Tuxedo DACS configuration from text file to binary, machine-readable form. The program *tmunloadcf* unloads (converts) the binary, machine-readable form back to a text file.

tmadmin

tmadmin is a command line utility that provides monitoring and control of the entire application. This Tuxedo client reads from and writes to the *BBL* running on the master machine to query and alter the distributed application.

qmadmin

qmadmin is a command line utility that provides monitoring and control of a disk qspace. This Tuxedo client creates, reads from, and writes to a qspace on a Tuxedo queue host machine.

Chapter 4: Detailed Design

This chapter describes the detailed design of the SAIC-developed DACS CSCs (non-COTS DACS) and includes the following topics:

- Data Flow Model
- Processing Units
- Database Description

Chapter 4: Detailed Design

This chapter introduces DACS servers, clients, and auxiliary programs that are part of the IDC software and have been developed and supplied by the PIDC. The purpose of this chapter is to describe the basic design of all SAIC-developed components. Operation of these components is described in [IDC6.5.2Rev0.1], and man pages describe all parameters that can be used to control and modify functions within the components. The first section, Data Flow Model, gives an overview of the interrelationships between the individual CSCs, which are described in detail in the Processing Units section.

DATA FLOW MODEL

In the context of Automatic Processing, the DACS includes CSCs for the following functions:

- Data monitoring
- Creation of pipeline processing sequences
- Centralized scheduling of the data monitoring servers
- Generalized execution and monitoring of Automatic Processing applications
- Centralized database updates
- Host-based routing of pipeline processing sequences by data source
- Automatic retries of failed pipeline sequences following system-level errors
- Interactive graphical presentation of all pipeline processing intervals including support for on-demand reprocessing

CSCs are also included for the operation of the DACS for Automatic Processing via several convenient GUI-based operator consoles.

In the context of Interactive Processing, the DACS includes CSCs for API level message passing between applications in the Interactive Processing CSCI as well as a GUI-based application for the monitoring of all interactive applications and messages within an interactive session. This latter CSC includes message-based demand execution, automatic execution, and user-assisted execution and termination of interactive applications within the session.

Figure 14 shows the data flow among the DACS CSCs for Automatic Processing. Tuxedo provides the reliable distributed processing infrastructure for DACS including reliable queuing, transactions, and process monitoring (bottom bar in Figure 14). DACS is controlled by the system operator through the centralized operator GUI *tuxpad* (a). Operator control includes complete DACS bootup or shutdown; bootup and shutdown on a machine basis, a process-group basis, or a process-server basis; control of the DACS scheduling system; and monitoring of Tuxedo queues. The DACS scheduling system is managed by *schedclient* (process 1), which is used to send commands[1] to the scheduling server, *scheduler* (process 2). The operational database is monitored by the DACS data monitor servers such as *tis_server* (process 3) in a recurring attempt to create processing intervals subject to data availability. Confirmation of sufficient data results in new interval information that is inserted into both the database and Tuxedo queues. The enqueues are either directly initiated by the data monitor server, or, optionally, the *interval_router* server can enqueue the interval data into one queue from a set of possible queues as a function of the interval name (process 4). System operators can use the *WorkFlow* application to monitor the progress of Automatic Processing (process 9), which renders database time interval states as colored bricks.

1. *tuxpad* is the most typical interface to *schedclient*.

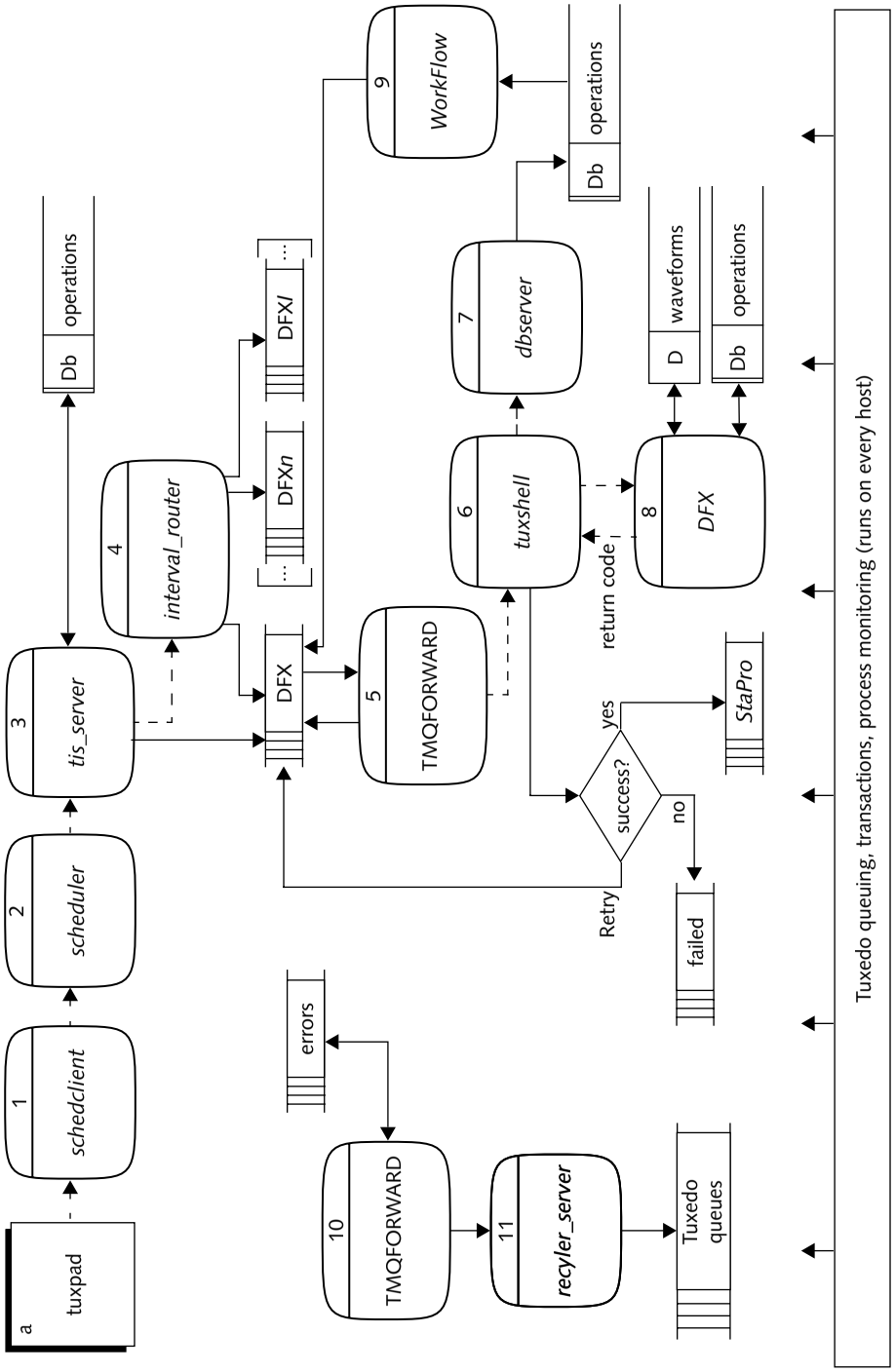


FIGURE 14. DATA FLOW OF DACS CSCS FOR AUTOMATIC PROCESSING

The Tuxedo queue forwarder, *TMQFORWARD*, passes the interval data to *tuxshell* as part of a service call (processes 5 and 6 in Figure 14 on page 50). The generalized processing server *tuxshell* calls one or more processing applications (for example, *DFX*) to send the processing interval to the desired/requested automatic processing task (process 8 in Figure 14 on page 50). *tuxshell* manages the execution of the processing task, handling a successful or failed run. Failed processing of an interval results in a transaction rollback of the queue message by *TMQFORWARD*. *TMQFORWARD* initiates reprocessing of the interval, which repeats the queue forwarding sequence (processes 5–8 in Figure 14 on page 50). Successful processing of an interval results in an enqueue of an updated message into another downstream Tuxedo queue (for example, *StaPro*) and a transactional commit of the original queue message dequeued by *TMQFORWARD*. The downstream Tuxedo queue manages the next step in the pipeline (processing sequence), which duplicates the queue forwarding sequence (processes 5–8 in Figure 14). *tuxshell* updates the interval data[2] in the database by sending an updated interval state to *dbserver*, which in turn issues the actual database update command to the ORACLE database (process 7 in Figure 14 on page 50). Queue intervals that failed due to system errors (for example, a machine crash) and have been directed to a system-wide error queue are automatically recycled back into the appropriate Tuxedo message queue by *recycler_server* (process 11 in Figure 14 on page 50).

Figure 15 shows the data flow among the DACS CSCs for Interactive Processing. Tuxedo provides the reliable message passing infrastructure for the DACS including reliable queuing and process monitoring (process 3). *libipc* provides the asynchronous message passing among the Interactive Tools within the Interactive Processing. This library is linked into all Interactive Processing clients (for example, *ARS* and *dman*) and is not explicitly listed in the figure. Actions within the interactive session are started by an analyst. The analyst either explicitly starts the analyst review station tool, *ARS* (process 2) or it is automatically started by *dman*, the DACS interactive session manager client (process 1).[3] Storing messages within a disk-based Tuxedo queue ensures that the messaging is asynchronous because the message send and receive are part of separate queuing operations and transac-

2. *dbserver* can update **interval**.*state* or **request**.*state*.

tions. Asynchronous messaging allows for one Interactive Tool (for example, *ARS*, process 2) to send a message to another Interactive Tool that is not currently running. *XkfDisplay* is used as an example in Figure 15, and similar control and data flow applies to other Interactive Tools. The *dman* client provides a demand execution feature, which starts an interactive client that is not already running and has a pending message (process 4). *dman* tracks all message traffic through Tuxedo IPC events, which are automatically broadcast to *dman* via the *libipc* message send and receive API calls that the Interactive Tools use. Access to Automatic Processing is provided for the purpose of interactive recall processing (process 2 and processes 5–7).[4] The *TMQFORWARD/tuxshell* configuration for managing Interactive Processing applications (processes 5–7) works in a similar but not identical manner with the DACS for Automatic Processing. In Interactive Processing, *TMQFORWARD* calls a *tuxshell* server within a transaction; however, the processing application status, success or fail, is sent back to the calling client via a *libipc* message (process 6). In addition, *tuxshell* does not attempt an **interval**.*state* update in the database because this processing is on-the-fly and is not represented as an interval in the database (the calling client, *ARS*, does not insert an interval into the database).

3. The interactive session can be managed by the *analyst_log* GUI application (not shown in Figure 15). This application manages analyst review by assigning blocks of time to analysts for analysis. This application can optionally start *dman*.
4. The label interactive recall processing (process 7 in Figure 15) refers generally to the various types of Automatic Processing that are used within Interactive Processing. These include interactive beaming (BOTF), interactive seismic recall (RSEISMO), interactive hydro recall (RHYDRO), and interactive auxiliary data request (IADR).

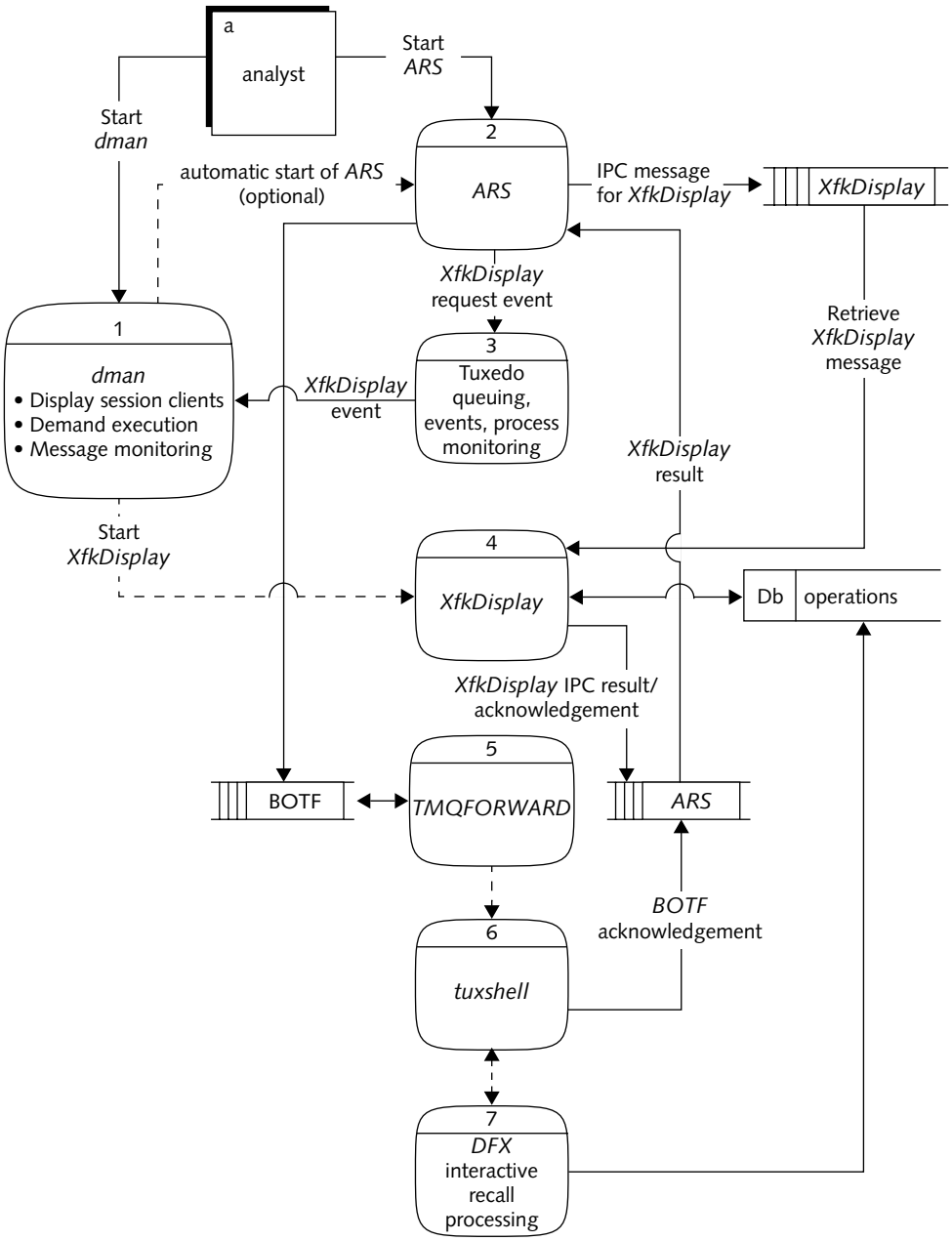


FIGURE 15. CONTROL AND DATA FLOW OF DACS CSCS FOR INTERACTIVE PROCESSING

PROCESSING UNITS

SAIC DACS CSCs consist of the following processing units:

- Data monitor servers: *tis_server*, *tiseg_server*, *ticron_server*, *tin_server*, and *WaveGet_server*
- *scheduler/schedclient*
- *tuxshell*
- *dbserver, interval_router, recycler_server*
- *WorkFlow, SendMessage, and ProcessInterval*
- *libipc, dman, and birdie*
- *tuxpad*, *operate_admin*, *schedule_it*, *quinfo*, and *msg_window*

The following paragraphs describe the design of these units, including any constraints or unusual features in the design. The logic of the software and any applicable procedural commands are also provided.

Data Monitor Servers

The DACS data monitor servers satisfy system requirements to monitor data availability to initiate automated pipeline processing as the availability criteria are met (Figure 16). The data monitor servers (*tis_server*, *tiseg_server*, *ticron_server*, *tin_server*, and *WaveGet_server*) share the following general design features:

- Initiate a processing cycle when called by *scheduler*.
- Apply the availability criteria using the database, and create or update data intervals inserting or updating rows in the **interval** or **request** table depending on the availability and timeliness of the data being assessed.
- Enqueue a message into a Tuxedo queue for 1) each new interval created with state *queued* and 2) each existing interval for which the state is updated from *skipped* to *queued* to initiate processing of an automated pipeline.

- Return an acknowledgment of completion of the processing cycle to *scheduler* by sending a `SETTIME` command to *scheduler* (perform an enqueue command to the *scheduler* command queue; see Figure 17 on page 56).

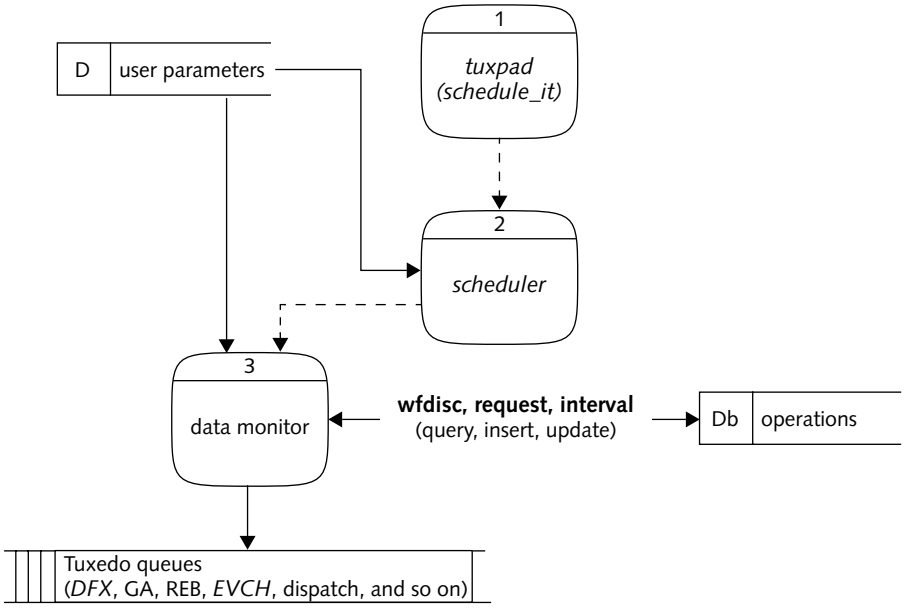


FIGURE 16. DATA MONITOR CONTEXT

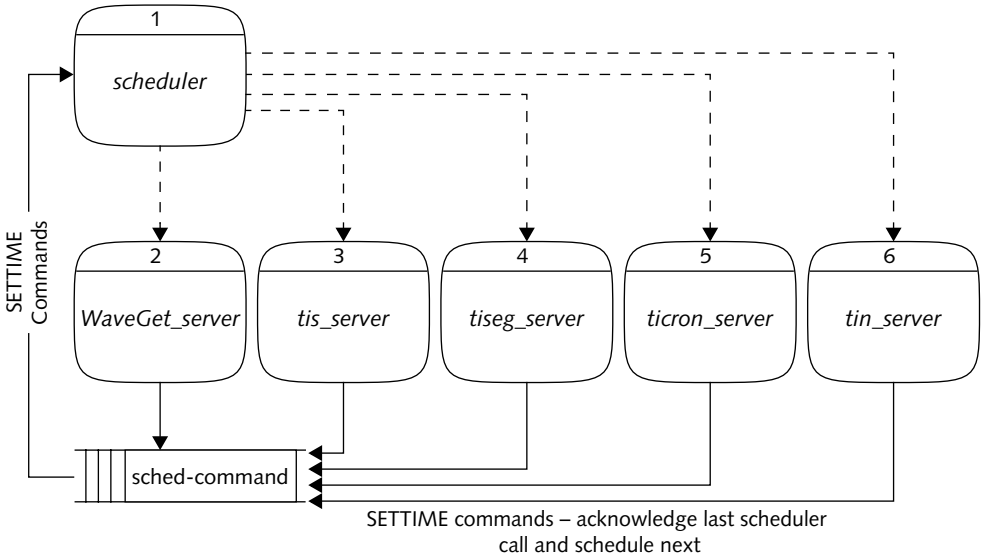


FIGURE 17. DATA MONITOR ACKNOWLEDGEMENT TO SCHEDULING SYSTEM

All of the data monitors are database applications, and all monitoring is based upon periodic polling of the database to check for availability based on varying criteria. Different data monitors are used to create different classes of intervals. User parameters define the queries used to check for the availability of data that each data monitor server is designed to assess. *tis_server* creates detection processing intervals based upon the availability of new continuous station data. *tiseg_server* creates detection processing intervals based upon the availability of new auxiliary seismic station data. *ticron_server* creates network processing intervals on a regular basis and of a fixed size. *tin_server* creates intervals of varying type based upon a trade-off between data availability and elapsed time. *WaveGet_server* initiates processing to acquire auxiliary station waveforms based upon requests for such data.

tis_server

tis_server creates and updates processing intervals of class *TI/S* for processing data from continuously transmitting stations. *tis_server* forms new candidate intervals based upon the timely arrival of new station data and updates existing intervals that were previously skipped due to incomplete or nonexistent station data.

The data flow for *tis_server* is shown in Figure 18. *tis_server* creates and updates intervals for all stations specified by the user parameters. The candidate interval check attempts to form a new interval for each station where the interval start time and end time are current. *tis_server* attempts to form a column of new intervals that would appear on the right side of the *WorkFlow* display (see Figure 27 on page 95). Candidate intervals are stored in a temporary, memory-based list during each *tis_server* cycle (M1). The candidate interval for each station is assessed for data coverage, and the interval is created if a sufficient percentage of overlapping station channels has arrived. The number of overlapping channels and percentage threshold is defined by the user parameters.

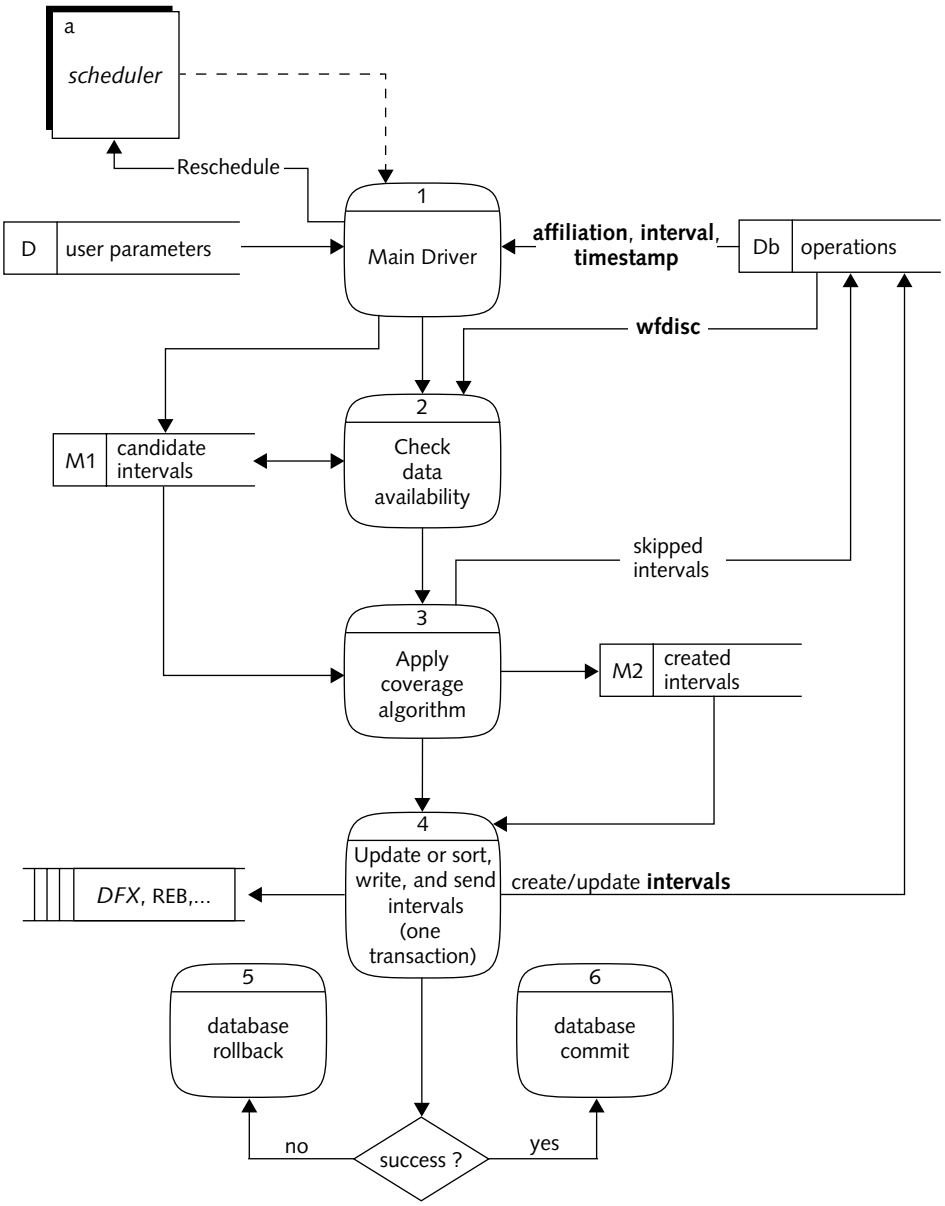


FIGURE 18. TIS_SERVER DATA FLOW

The data coverage algorithm accumulates the number of seconds of overlapping channels for each station and then calculates a coverage percentage. The coverage percentage is compared to the user-specified threshold value, and if sufficient data are found, a new interval is created and stored in memory (M2 in Figure 18). The new interval *state* is set to `queued`. A message containing information about the interval is enqueued into a Tuxedo queue that initiates pipeline processing. If the threshold is not exceeded, **interval**.*state* is set to `skipped`, and the interval is not queued for processing.[5] Figure 19 shows the logic used to form intervals for current data and check for skipped data. Candidate intervals of user-specified length are formed by *tis_server* between the end of the last existing time interval in the **interval** table (yellow brick, see "Current Data" in Figure 19) and the end of the newest data record in the **wfdisc** table (black bars, see "Current Data" in Figure 19) for a particular station (white brick candidate intervals, see "Current Data" in Figure 19). These intervals are inserted into the **interval** table by *tis_server* (see "Current Data" in Figure 19).

5. A skipped interval is created only if a queued interval exists (or has been confirmed) later in time than the skipped interval. That is, a skipped interval is never a leading interval. As a result, a skipped interval following a station outage only appears after the station resumes transmitting data, which results in one or more new queued intervals.

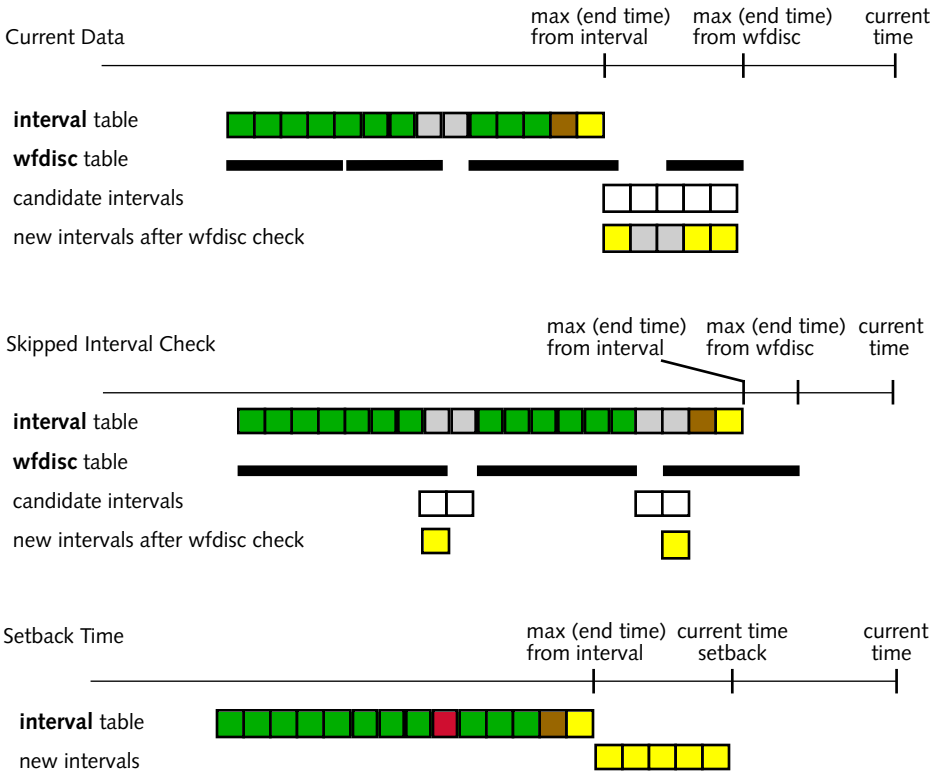


FIGURE 19. CURRENT DATA AND SKIPPED INTERVAL CHECKS

Candidate intervals that were not enqueued for processing by *tis_server* because the threshold value was not exceeded are known as "skipped" intervals. However, late-arriving data may complete an interval and *tis_server* may check the data contents of all skipped intervals (light gray bricks, see "Skipped Interval Check" in Figure 19) to see if enough data have been received to surpass the threshold percentage (black bars, see "Skipped Interval Check" in Figure 19). If a skipped interval for which the threshold percentage has been exceeded is found, **interval**.*state* is updated to `queued`, (yellow bricks–"new intervals after wfdisc check," see "Skipped Interval Check" in Figure 19) and a corresponding message is enqueued into a Tuxedo queue.

tis_server can create new intervals or update previously skipped intervals based only upon the addition of other intervals in the database. Therefore, *tis_server* is not necessarily dependent on wfdiscs. More generally, *tis_server* requires start time and end time data. The start time and end time could be related to database wfdiscs or just as easily to the start time and end time of database intervals. Therefore, it is possible to specify query parameters that are entirely based upon the **interval** table whereby *tis_server* forms new intervals based upon the progress of other related intervals. This generalized use of *tis_server* is employed in a number of cases to form pipeline processing sequences based upon the existence of specific interval states within a specified range of time. The design of *tis_server* addresses a number of complexities specifically related to continuous station data transmission (wfdisc-based monitoring). Therefore, the more general interval-based monitoring uses of *tis_server* exercise a relatively small percentage of the server's features.

tiseg_server

tiseg_server creates intervals of the class *TI/B* that correspond to relatively short segments of irregular duration from auxiliary seismic stations. The created intervals are enqueued into a Tuxedo queue to initiate detection and station processing. *tiseg_server* periodically checks the **wfdisc** table for new entries originating from seismic stations. Each auxiliary seismic station has a designated monitor channel that serves as the time reference channel for forming the *TI/B* intervals in the **interval** table. Complete (`queued`) intervals are formed in the **interval** table when the monitor channel is found along with all other expected channels (Figure 20). Incomplete (`partial`) intervals are formed when the monitor channel is found in the absence of a specified minimum number of related station channels. Partial intervals in the **interval** table are completed (updated to `queued`) when the minimum number of missing channels can be found within a user-specified elapsed time period.

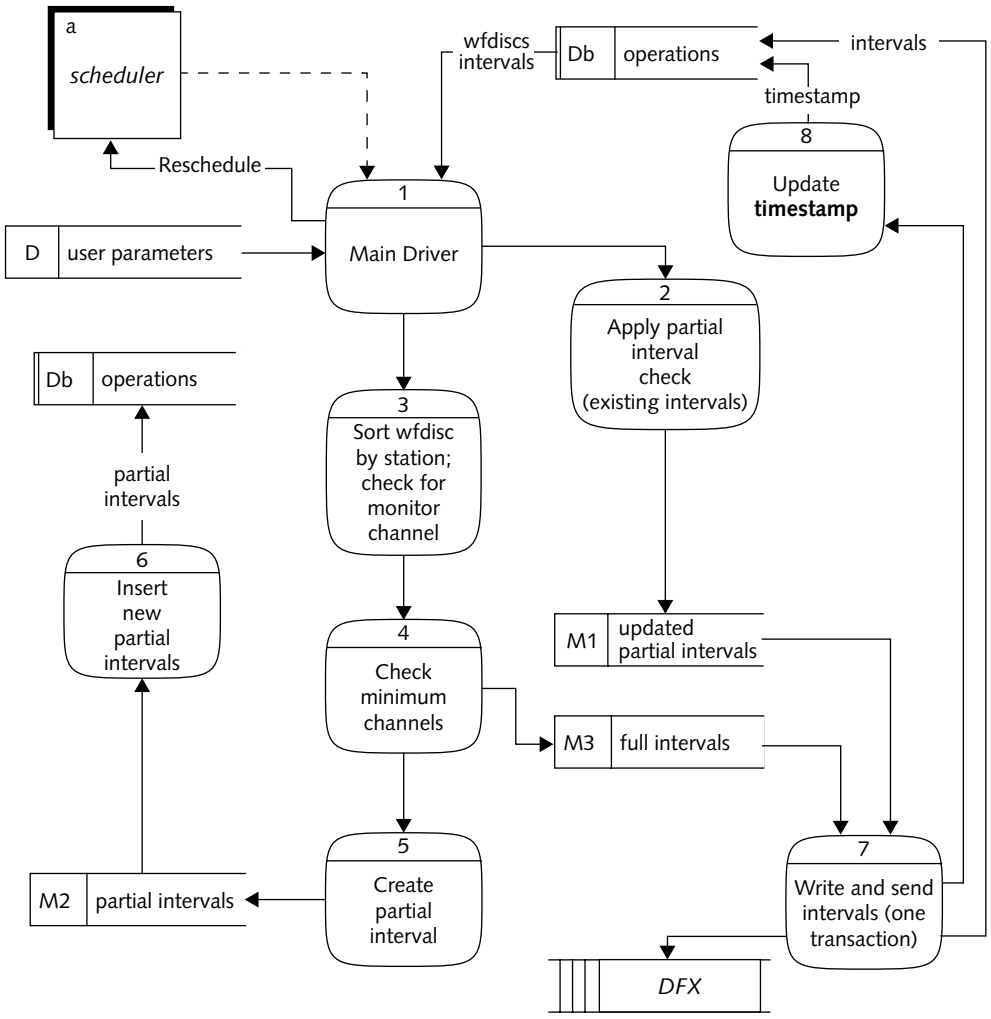


FIGURE 20. TISEG_SERVER DATA FLOW

ticron_server

ticron_server creates fixed-length intervals of the class *TI/N* based on a fixed elapsed time (setback) prior to the current real time. Created intervals are inserted into the **interval** table and a Tuxedo queue to initiate network processing (Figure 21). The length of the intervals is nominally set to 20 minutes, but this parameter and other parameters are user configurable.

Network processing is performed several times at successively greater time delays from the current time to produce the various bulletin products of the IDC. To maintain the delay in processing, a setback time is used. The bottom portion of Figure 19 on page 60 shows the setback criterion used by *ticron_server* (yellow bricks–"new intervals," see "Setback Time" in Figure 19 on page 60). The effect of applying this criterion is that network processing in the SEL1, SEL2, and SEL3 pipelines maintains constant delays (currently 1 hour 20 min, 5 hours 20 min, and 11 hours 20 min, respectively) relative to the current time.

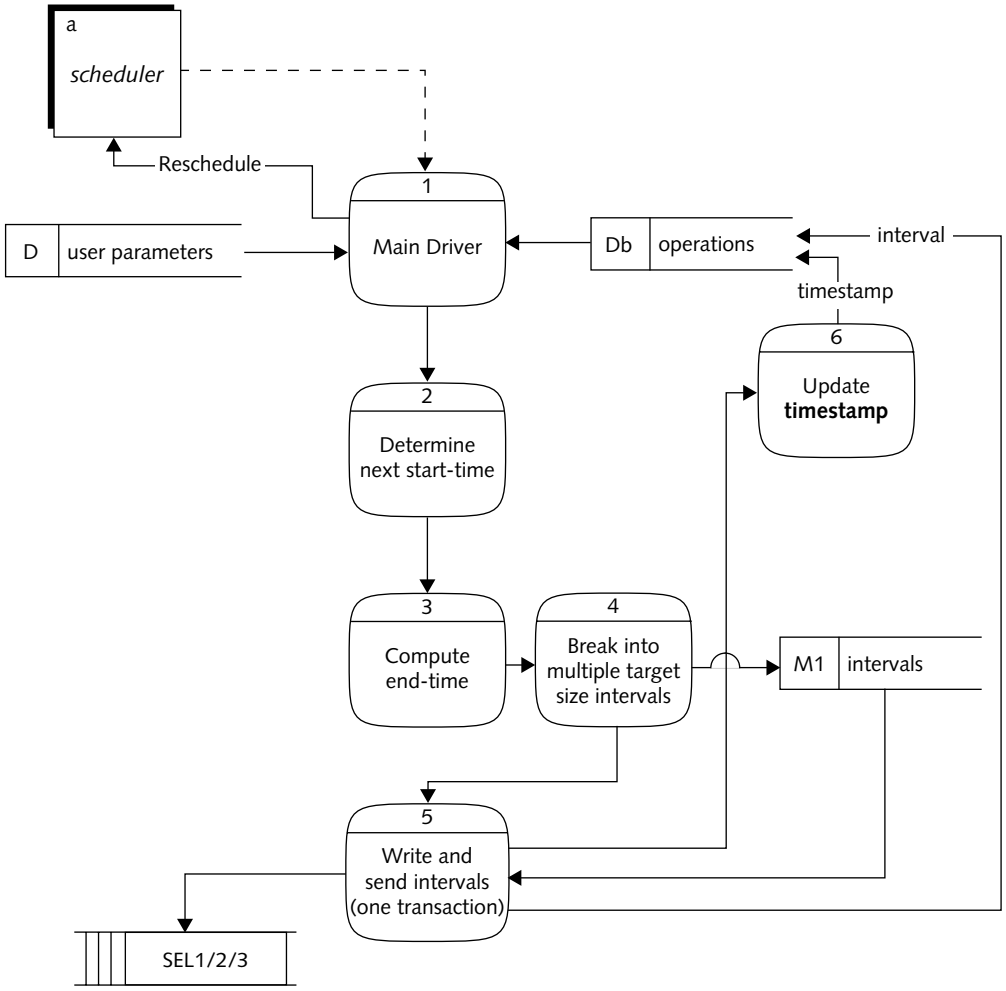


FIGURE 21. TICRON_SERVER DATA FLOW

tin_server

tin_server creates intervals based upon a trade-off between data availability and elapsed time. Intervals of class *TI/N*[6] are inserted into the **interval** table, and the interval information is enqueued into a Tuxedo queue to initiate pipeline processing. The data availability criterion is based upon the number of completed intervals

for a given class or group of processing (processes 5–7 in Figure 22). The processing class or group is flexible in that *tin_server* exclusively relies on an integer returned from a user-defined SQL query. Thus *tin_server* is not concerned with network or station affiliations, and the user-defined data count query must map the completion status of the monitored station set or group to an integer number. A dedicated instance of *tin_server* is required for each processing group or class (for example, three hydroacoustic groups require three dedicated *tin_server* instances).

The data availability versus time criteria are based on two user-defined value arrays of equal dimension. These arrays define the minimum number of data counts or completions acceptable at a time elapsed relative to present time and the end time of the last interval created. In general, the data count thresholds reduce and/or the data completeness threshold is relaxed as elapsed time increases. If sufficient data are confirmed, a complete interval is created and the interval information is enqueued into a queue. If insufficient or no data are available after a defined amount of time, a skipped interval is created. The end time of the created interval, whether complete or skipped, defines the start time for the next interval's elapsed time measurement. The updating of skipped intervals is based upon a user-defined SQL query. *tin_server* does not supply time values for substitution in the SQL query. Skipped intervals returned from the query are updated to complete intervals, and then enqueued into a queue (process 2).[7]

6. The IDC software uses *tin_server* to create intervals for Hydroacoustic Azimuth Estimation, which are labeled with the class *HAE*. Explicit classes and states of intervals are configurable for each data monitor. This document lists the generic names which coincide often, but not always, with explicit names.
7. There are few or no requirements for skipped interval processing for *tin_server*. The creation of skipped intervals is intended primarily to keep interval creation current relative to present time, thereby avoiding interval gaps or the stalling of interval creation due to delays or failure of the processing that is monitored by *tin_server*.

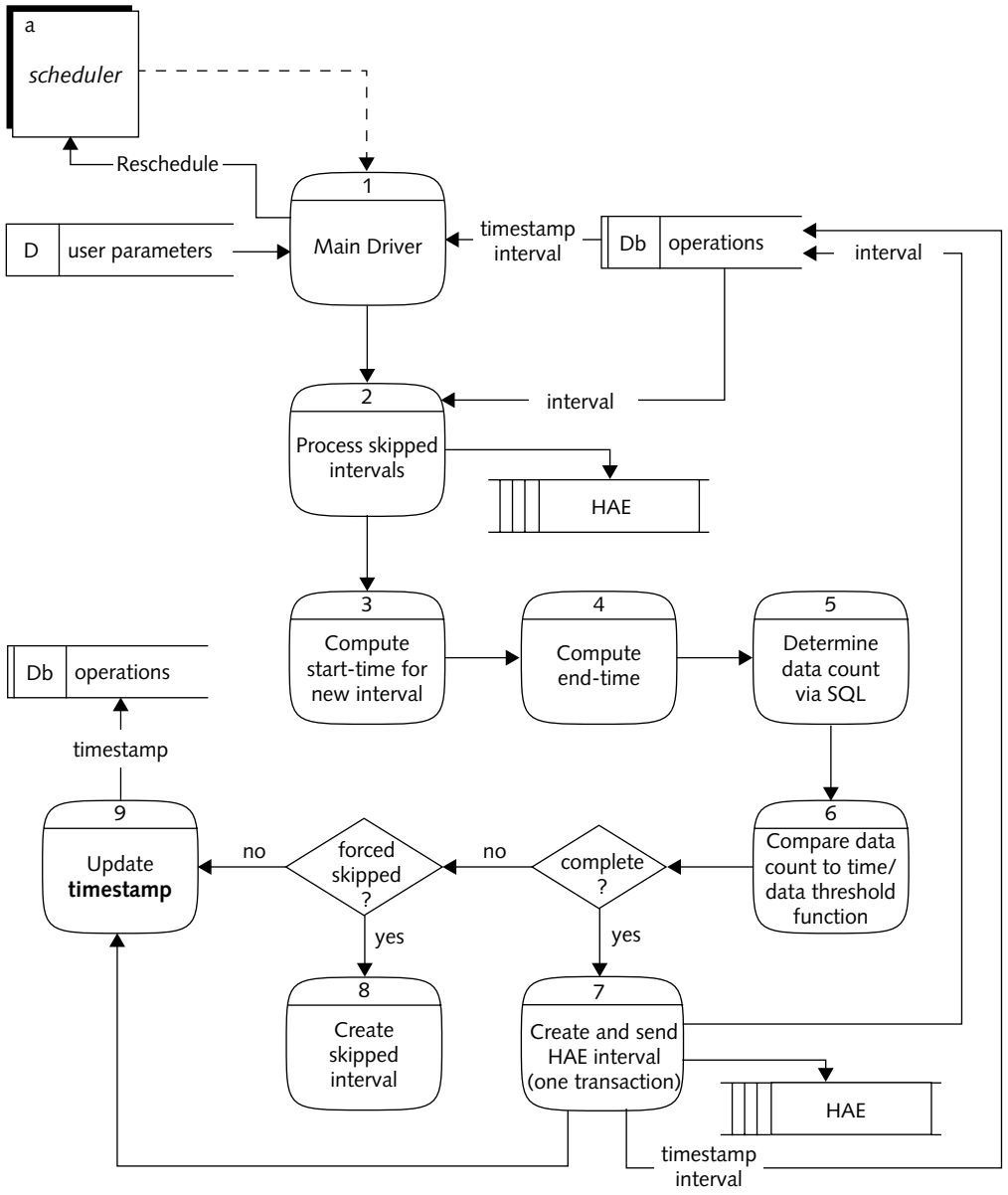


FIGURE 22. TIN_SERVER DATA FLOW

WaveGet_server

WaveGet_server is a data monitor server that polls the **request** table for auxiliary-station-waveform requests and initiates actions to acquire the requested waveforms. The actions include IPC message enqueues into one or more Tuxedo queues and the updating of the state of the revised requests in the database. The IPC messages consist of the updated request information. The enqueued messages initiate pipeline processing that ultimately results in auxiliary waveform being requested by the Retrieve Subsystem. *WaveGet_server* processes both new requests and previous requests that have failed to result in successful auxiliary waveform acquisition.

WaveGet_server provides standard mode and archival mode processing. Standard mode processing operates on incomplete requests for data. Archival mode processing operates on requests for which too many retrieval attempts have failed or too much time has elapsed.

In standard mode processing, *WaveGet_server* sorts all active requests for data by four different criteria. The first sort is by priority of request, the second is by transfer method, the third is by station, and the fourth is by time.

WaveGet_server prioritizes the requests based upon a list of priority names defined by the user parameters. The priority names define different request types, and within each priority level the requests are grouped by transfer method. Within a transfer method, the requests are sorted by station and by time. After all active requests are sorted, one IPC message per request is enqueued into the configured Tuxedo queue (process 4 in Figure 23).

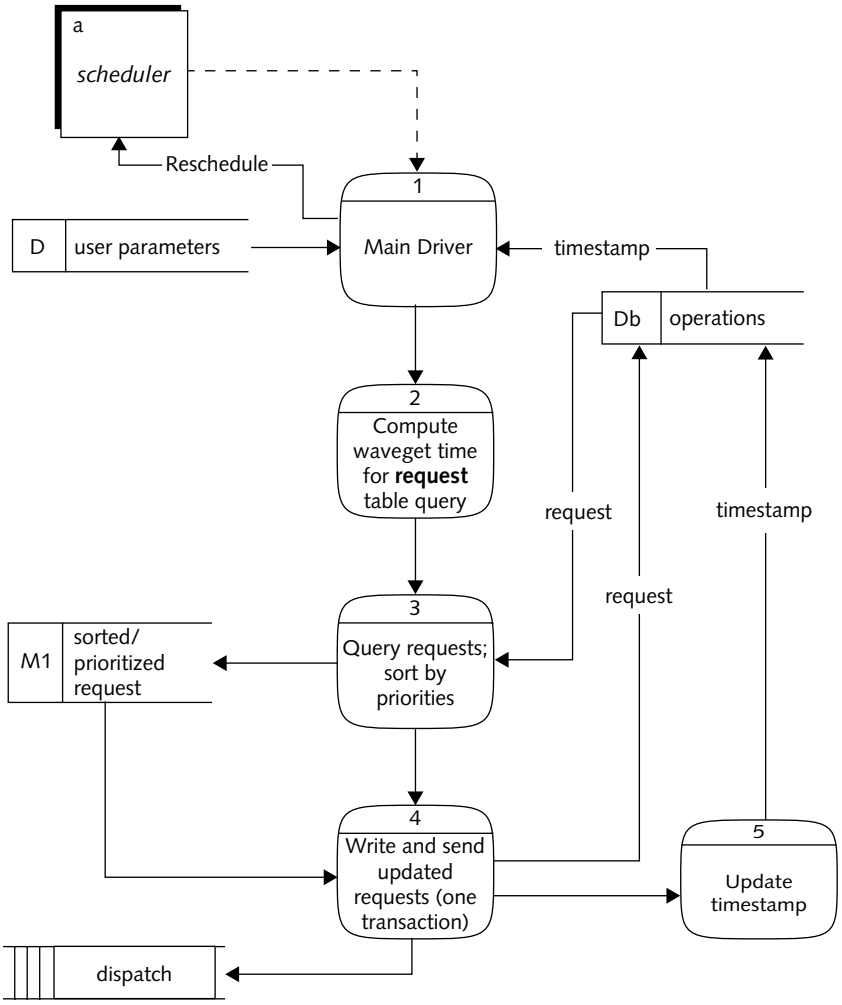


FIGURE 23. WAVEGET_SERVER DATA FLOW

WaveGet_server manages the retry of previous failed requests. Failures are detected by the DACS and recorded in the **request** table.[8] *WaveGet_server* reprocesses previous failed attempts after a small time interval has elapsed.

In archival mode, *WaveGet_server* changes the state of selected entries in the **request** table. The intent is to change the state of requests that have either too many failures or are too old. The new state both prevents *WaveGet_server* standard mode from considering these requests and provides a clear indication to an operator that the request is no longer being considered by *WaveGet_server*.

Input/Processing/Output

tis_server

Figure 18 on page 58 shows data and processing flow for *tis_server*. *tis_server* receives input from user-defined parameter files, the database, and the *scheduler* server. The parameter files specify all processing details for a given instance of the data monitor server. Details include database account, station names, database queries, and interval coverage threshold values. The user parameters are used to construct the recurring database queries to check or monitor the availability of new station data. Initial database input to *tis_server* includes station and network affiliations used to build a complete station, site, and channel table for all monitored stations.

tis_server creates and updates intervals for processing data from continuously transmitting stations. *tis_server* forms new candidate intervals based upon the timely arrival of new station data and updates existing intervals that were previously skipped due to incomplete or nonexistent station data.

tis_server generates output to log files, the database, Tuxedo queues, and the *scheduler* server. Output to the database includes new intervals, be they incomplete (**interval**.*state* = `skipped`) or complete (**interval**.*state* = `queued`). Updates to the database include previously skipped intervals updated to queued intervals following the verification of newly arrived data. *tis_server* also optionally supports

8. *WaveGet_server* detects Retrieve Subsystem request retrieve failures by querying the **request**.*state* and **request**.*statecount* in the database. Depending on the state and number of failed requests (value of **request**.*statecount*), *WaveGet_server* determines whether subsequent requests should be made or the state should be updated to failed to terminate the request and eliminate it from consideration in future *WaveGet_server* invocations.

output to the **timestamp** table to track interval creation by station. However, in practice, the **timestamp** updates are carried out by database triggers that update this information based upon updates to the **wfdisc** table. (This performance optimization can be considered part of the *tis_server* design, but its implementation is external to *tis_server*). Upon interval creation, *tis_server* enqueues a message containing the interval information into a Tuxedo queue for initiation of a pipeline processing sequence on the time interval. *tis_server* completes its interval creation cycle by sending an acknowledgement `SETTIME` command to the *scheduler* server, which results in rescheduling for the next *tis_server* service call.

tiseg_server

Figure 20 on page 62 shows data and processing flow for *tiseg_server*. *tiseg_server* receives input from user-defined parameter files, the database, and the *scheduler* server. The parameter files specify all processing details for a given instance of the data monitor server. Details include database account, auxiliary network, database queries, and station- and time-based interval coverage values. The user parameters are used to construct the recurring database queries to check or monitor the availability of new station data. Initial database input to *tiseg_server* includes an auxiliary network, which is used to build a complete station, site, and channel table for all monitored auxiliary stations.

tiseg_server first carries out partial interval processing (process 2 in Figure 20 on page 62). An attempt is made to declare each partial interval complete, querying the database for data availability of the remaining channels for the auxiliary station in question. Data completeness is defined by all remaining channels or some subset subject to user-defined parameters. When the minimum number of auxiliary station channels is confirmed, **interval**.*state* is updated to *queued* and the interval information is enqueued to a Tuxedo queue (for example, *DFX* queue) to initiate pipeline processing (process 7 in Figure 20 on page 62).

The second and primary processing task of *tiseg_server* is the interval creation algorithm whereby complete and partial intervals are created. The interval creation algorithm includes a sort of all **wfdisc** rows by station names (process 3 in Figure 20 on page 62) to organize interval creation and processing in station lexicographic

order. The availability of waveforms on the user-defined monitor channel results in the creation of a *TI/B* interval. The interval is considered only partial if the monitor channel is not joined by the minimum number of affiliated channels for the auxiliary station (process 5 in Figure 20 on page 62) in a check of criteria identical to the partial interval check (process 2 in Figure 20 on page 62). If the monitor channel is joined by the minimum number of affiliated channels for the auxiliary station, a new row with state *queued* is inserted into the **interval** table, and the interval information is enqueued into a Tuxedo queue (process 7 in Figure 20 on page 62).

tiseg_server generates output to log files, the database, Tuxedo queues, and the *scheduler* server. Output to the database includes new intervals, both incomplete (**interval**.*state* = `partial`) or complete (**interval**.*state* = `queued`). Updates to the database include previously partial intervals updated to queued intervals following the verification of newly arrived data. *tiseg_server* updates the **timestamp** table with the current time to record the most recent time of a successful interval creation by *tiseg_server*. Upon interval creation, *tiseg_server* enqueues a message containing the interval information into a Tuxedo queue for initiation of a pipeline processing sequence on the interval. *tiseg_server* completes its interval creation cycle by sending an acknowledgement `SETTIME` command to the *scheduler* server, which results in rescheduling for the next *tiseg_server* service call.

ticron_server

Figure 21 on page 64 shows data and processing flow for *ticron_server*. *ticron_server* receives input from user-defined parameter files, the database, and the *scheduler* server. The parameter files specify all processing details for a given instance of the data monitor server. Details include database account, class and size of target intervals to be created (for example, SEL1, 20 minutes), database queries, and time-based interval creation values (for example, the setback time). The user parameters are used to construct the recurring database queries to determine the time and duration of the last interval class created. Initial database input to *ticron_server* includes timestamp and interval information, which is used to build new time interval(s) depending on when the last interval was created and the current time.

ticron_server processing is straightforward and creates intervals as a function of time. The *ticron_server* interval creation algorithm includes determination of a start time for the next interval it will create. This start time is a function of the most recent end time of the last created interval/value noted optionally in the **timestamp** table (process 2 in Figure 21 on page 64). Associated end times for each interval are computed as a function of the target interval size and a user-defined time set-back value (process 3 in Figure 20 on page 62). One or more intervals are created by *ticron_server* depending on whether the computed new interval of time exceeds the target interval length (process 4 in Figure 21 on page 64). Completed interval(s) are written to the database and then enqueued into a Tuxedo queue to initiate pipeline processing of the intervals.

ticron_server generates output to log files, the database, Tuxedo queues, and the *scheduler* server. Output to the database includes new **intervals** and updates to the **timestamp** table. Upon interval creation, *ticron_server* enqueues a message containing interval information into a Tuxedo queue for initiation of a pipeline processing sequence on the interval. *ticron_server* completes its interval creation cycle by sending an acknowledgement `SETTIME` command to the *scheduler* server, which results in rescheduling for the next *ticron_server* service call.

tin_server

Figure 22 on page 66 shows data and processing flow for *tin_server*. *tin_server* receives input from user-defined parameter files, the database, and the *scheduler* server. The parameter files specify all processing details for a given instance of the data monitor server. Details include database account, class, name, and size of target intervals to be created (for example, HAE, WAKE_GRP, 10 minutes), database queries, and arrays of time and data count values. These values define the time/data threshold function for interval creation. The user parameters are used to construct the recurring database queries to determine the time and duration of the last interval created so that the start time and end time of the next interval can be established. Initial database input to *tin_server* includes timestamp and interval information used to establish the times of the next interval.

The *tin_server* interval creation algorithm creates a timely or current interval. The interval is complete if a sufficient number of data counts versus elapsed time can be confirmed. The interval is unresolved if insufficient data counts are present but elapsed time has not run out. The interval is incomplete if insufficient data counts cannot be confirmed following a maximum user-defined time lapse. All time-based comparisons are relative to the present time and the end time of the last interval created. *tin_server* computes the start time for the current interval as a function of the last interval created, a value from the **timestamp** table, and a user-defined look-back value (process 3 in Figure 22 on page 66). The **timestamp** value and lookback value are generally only relevant if no previous intervals exist in the database such as the case upon system initialization (when a new system is run for the first time). The end time is computed as a function of the user-defined values for target interval size and time boundary alignment. The latter feature allows for interval creation that can be snapped to a timeline grid such that intervals fall evenly on the hour/ the selected minute interval (process 4 in Figure 22 on page 66). Having established the candidate interval start and end time, the interval creation algorithm proceeds to confirm the required data counts as a function of time (as described above and shown in processes 5 and 6 in Figure 22 on page 66). The data count query is user-defined and is usually targeted at a logical processing group such as a network of seismic stations or a group of hydroacoustic sensors. Complete intervals are created along with an enqueue into a Tuxedo queue as one logical transaction (process 7 in Figure 22 on page 66). Following a successful complete interval creation and enqueue, the end time of the interval is recorded in the **timestamp** table (process 9 in Figure 22 on page 66). Incomplete intervals are created absent an enqueue (process 8 in Figure 22 on page 66).

tin_server generates output to log files, the database, Tuxedo queues, and the *scheduler* server. Output to the database includes the complete and incomplete **intervals** and **timestamp** table updates. Upon interval creation, *tin_server* queues the time interval information to a Tuxedo queue for initiation of a pipeline processing sequence on the time interval. *tin_server* completes its interval creation cycle by sending an acknowledgement `SETTIME` command to the *scheduler* server, which results in rescheduling for the next *tin_server* service call.

WaveGet_server

Figure 23 on page 68 shows data and processing flow for *WaveGet_server*. *WaveGet_server* receives input from user-defined parameter files, the database, and the *scheduler* server. The parameter files specify all processing details for a given instance of the data monitor server. Details include database account, state names used for query and update of the **request** table, database queries, and values for sorting and managing the requests. The user parameters are used to construct the recurring database queries to determine if any requests should be passed to the messaging system or if any requests should be declared failed and aborted (so that no further data requests are attempted).

In standard mode processing *WaveGet_server* considers recent requests subject to the three factors: maximum lookback, current time, and time of last run. Determination of the time interval is a function of a user-specified maximum lookback, current time, and the most recent run of the *WaveGet_server* cycle, which is recorded in the **timestamp** table (process 2 in Figure 23 on page 68). The time interval or time period of interest is inserted into a user-specified request query, which retrieves all requests (process 3 in Figure 23 on page 68). The user-specified query is purposely flexible so that any practical query filters or clauses can be applied. The retrieved requests are sorted according to four search criteria including a user-specified priority and the request's transfer method, name and time. The sorted list is recorded in a memory-based list and is the central data structure for all server operations (process 3 and M1 in Figure 23 on page 68). The sorted list is pruned of any request names that are not defined in the user-defined list of station names. The pruning involves updating the request states to a user-specified ignore state, which removes the request from further consideration. The sorted list of requests is updated in the database and sent to a Tuxedo queue as one global transaction (processes 4 and 5 in Figure 23 on page 68).

In archival mode processing *WaveGet_server* will set **request**.*state* = `failed` for all old requests that have not resulted in successful auxiliary waveform acquisition within a user-specified time lookback and/or have failed an excessive number of times.

WaveGet_server generates output to log files, the database, Tuxedo queues, and the *scheduler* server. Output to the database includes updates to the **request** table and **timestamp** table. **request** table updates to *state* `queued` are coupled with enqueues of the request information to a Tuxedo queue. The enqueue initiates the pipeline processing sequence to retrieve the requested auxiliary waveform. *WaveGet_server* completes its processing cycle by sending an acknowledgement `SETTIME` command to the *scheduler* server, which results in rescheduling for the next *WaveGet_server* service call.

Control

Tuxedo boots, monitors, and shuts down the data monitor servers: *tis_server, tiseg_server, ticron_server, tin_server,* and *WaveGet_server.* Server booting is either initiated by an operator directly using Tuxedo administrative commands or indirectly via *tuxpad*, or, automatically via Tuxedo server monitoring. During Tuxedo server monitoring servers are automatically restarted upon any failure. An operator initiates the server shut down.

Control of the data monitor server function is largely defined by the user parameters. However, the scheduling system enables an operator to start the data monitor service on demand such that a data monitor cycle can be called at any time, otherwise the data monitor service is automatically called by the scheduling system on a recurring scheduled basis. In addition, the same interface allows for stalling and unstalling data monitor service requests, which results in the ability to control whether or not a data monitor server is active and able to initiate interval creation.

Interfaces

The data monitor servers are database applications, which receive input data from the database, then exchange or store that data in internal data structures for various types of interval- creation algorithms. The detailed process or control sequencing within each data monitor, including internal interfaces, is shown in each of the data monitor server data flow diagrams (Figures 18–23).

Error States

The data monitor servers can handle three primary failure modes: a spontaneous data monitor server crash, a database server failure, and a Tuxedo queuing failure. Attempts are made to automatically recover from each failure mode.

Spontaneous data monitor server crashing normally results from a previously unexercised program defect or a system resource limit. Tuxedo automatically restarts the data monitor servers upon server failure. Server failures due to system resource limitations (for example, swap or virtual memory exceeded) can be more easily recovered from than those from program defects because such a resource error may be transient or resolved by operator intervention. In this case the failure recovery is automatic for the data monitor server. Server failures due to a previously unknown program defect are typically more problematic because although the program reboot is automatic, the program defect is often repeated, resulting in an endless server reboot cycle.

The data monitor servers accommodate a variety of database server error conditions. If the database server is unavailable, the data monitor server attempts to reconnect for a maximum number of times during the current interval creation cycle before giving up. This cycle is repeated during subsequent calls to the data monitor server in an attempt to reconnect to the database server. In this scenario, the data monitor servers never crash or terminate due to database server downtime. General database query, insert, or update errors are handled via an attempt to rollback as much of the interval creation cycle work, or progress as much as possible prior to ending the current interval creation cycle. Included in this error state processing is an attempt to keep Tuxedo queue inserts and database inserts or updates as one transaction such that the database operation(s) are not committed until the Tuxedo enqueue(s) are successful. This is shown in all of the data monitor data flow diagrams (Figures 18–23). Errors for all database failures are logged to the data monitor log files.

scheduler/schedclient

scheduler and *schedclient* support the DACS scheduling system. *scheduler* satisfies the requirement for a centralized server for automatic data monitor calls, and *schedclient* satisfies the requirement for a tool for the centralized management of the scheduling system. The DACS data monitor application servers (for example, *tis_server*, *WaveGet_server*) await service calls from *scheduler* to carry out their data monitoring service and return acknowledgments to *scheduler* following completion of their service cycle. The scheduling system was designed to be fault tolerant. To achieve this objective the system is based upon the reliable Tuxedo disk queuing system.

The principal design decision involved the selection of either the database or the Tuxedo queuing system as a stable storage resource. The database is a single point of failure. The Tuxedo queuing system includes an automatic backup queuing with some limitations. The state of the primary queuing system is frozen until recovery by operator intervention. Such a scenario works for the DACS Automatic Processing software where new interval creation and processing proceeds by using the backup DACS qspace even though unfinished intervals are trapped in the primary qspace until the primary queuing system is restored. This scenario is not sufficient for the scheduling system because the scheduler state is frozen during queuing system failure, and there is one and only one scheduling system state. As such, the Tuxedo queuing system is also a single point of failure for the scheduling system. After weighing various trade-offs, a decision was made to base the scheduling system on the Tuxedo queuing system. Justifications for this decision included an implementation that appeared to be more straightforward and consistent with the rest of the Tuxedo-based DACS implementation and some promise for achieving seamless fault tolerant in the future.[9]

9. Hardware solutions such as dual ported disk drives have been shown to provide seamless fault tolerance within a Tuxedo queuing system.

Input/Processing/Output

Figure 24 shows the design of the fault-tolerant scheduling system. The sequenced queuing, transaction, and execution steps are numbered. The Tuxedo reliable queuing system provides the foundation for the reliable scheduling system. The queuing system consists of the built-in Tuxedo forwarding servers, *TMQFORWARD*, as well as queues Q1 (schedule), Q2 (sched-command), and Q3 (sched-result), the scheduler state, command, and result queues respectively. *scheduler* and *schedclient* input, output, and control flow are also shown in the figure. However, the figure does not show that both the *scheduler* servers and *schedclient* receive input from user parameters (via *libpar*).

The scheduler state consists of the table of scheduled services and their next due time and other global state (for example, *kick* state). When this due time is equal to current time, *scheduler* issues a service call to a server advertising the required service. These services are typically advertised by data monitors. For example, *tis_server* advertises services *tis*, *tis-late*, *tis-verylate* and others. The state table is encapsulated in one Tuxedo queue element that is reliably maintained in the state queue, Q1. The queue structure is based upon a Tuxedo Fielded Markup Language (FML) message.

The state queue must be seeded with an initial scheduler table at least the first time the system is started. This is accomplished by the `schedclient init` command. This command empties the state queue, if necessary, and then enqueues the initial state into the state queue, (step 1[10]). Subsequent system restarts can optionally issue another `init` command upon system bootup, or they can choose to pick up exactly where the system left off, because the last scheduler state remains in the state queue.

10. *schedclient* shuts down the *TMQFORWARD* server prior to dequeuing the scheduler state from the state queue and then reboots the *TMQFORWARD* after enqueuing the new initial state into the scheduler state to complete the reset of the scheduling system. The *TMQFORWARD* server is shut down and started through Tuxedo *tmadmin* commands that are generated and issued by *schedclient*. The *TMQFORWARD* management is necessary to avoid race conditions whereby *TMQFORWARD* might dequeue the scheduler state before *schedclient,* which would result in two (or more) scheduler states; this would manifest in repeated and possibly conflicting scheduling calls to the data monitor servers.

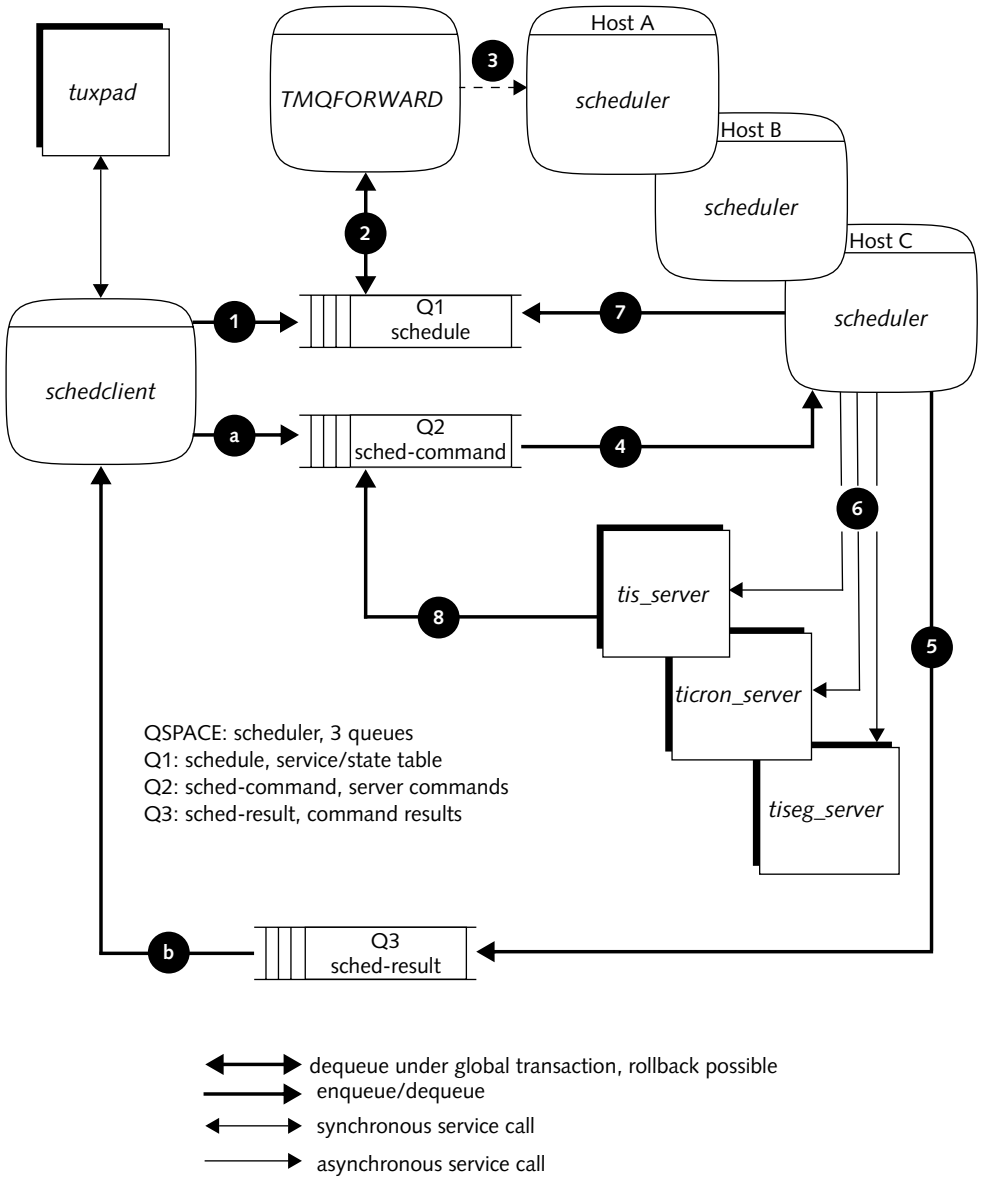


FIGURE 24. SCHEDULING SYSTEM DATA FLOW

The state queue consists of one and only one queue element, the scheduler state; this is the key to the fault-tolerant design. *TMQFORWARD* starts a transaction, (step 2), and then dequeues and forwards the queue message (the state) to one of the *scheduler* servers running on any of several hosts, (step 3). It does not matter which *scheduler* server receives the call because all servers are equally stateless until they are passed state within the global transaction.

If one or more commands exist in the command queue they are dequeued, (step 4), and applied to the scheduler state, resulting in an updated state. This updated state is requeued into the state queue, (step 7). At this point the state queue technically has two queue elements in it: the previous and the updated scheduler state. However, neither queue element is visible to the rest of the system until the global transaction is resolved by either commit or rollback, after which only one queue element will remain in the state queue. If *scheduler* returns success to *TMQFORWARD*, (step 3), following success of the updated requeue, (step 7), *TMQFORWARD* will commit the global transaction. This commit operation results in the commits of the original dequeue operation (step 2), after commits for the command(s) dequeued, (step 4), any results enqueued, (step 5), and the enqueue of the updated state, (step 7). Otherwise, if *scheduler* returns fail to *TMQFORWARD*, (step 3), *TMQFORWARD* rollsback the global transaction. This rollback operation negates all queuing operations including any dequeues from the command queue, (step 4), enqueues to the result queue, (step 5), requeues to the state queue, (step 7) and the original dequeue from the state queue, (step 2).

Prior to *scheduler* returning success to *TMQFORWARD* and final transaction commit (step 7), data monitor servers are called for all services that are at or past this scheduled time, (step 6). The data monitor service call is asynchronous and cannot be rolled back; therefore it is not considered part of the global transaction. In practice, this limitation does not present a problem because the function of the scheduling system is to call the data monitor servers on schedule. Failure of a data monitor service is outside the scope of the scheduling system design. The best form of error handling is a repeated attempt to call the data monitor server. As such, *scheduler* always schedules a subsequent call to the data monitor service immediately after the service call. This worst case schedule time is typically set beyond the time the service would next normally be called and is tunable via user

parameters. A successful data monitor service call completes with an acknowledgment SETTIME command, (step 8 in Figure 24 on page 79), enqueued into the command queue. This acknowledgment command results in an update of the next scheduled time to call this data monitor service.

scheduler commands and results pass through the command and result queues. The results of most commands are simply a boolean success or fail. The *show* command is an exception where *scheduler* returns the human readable listing of scheduled services. *scheduler* commands and results are matched by the Tuxedo queue-based *correlation identifier* that is used by both *scheduler* and *schedclient*. *schedclient* polls the result queue, (step b in Figure 24 on page 79), and searches for the matching result of the command that was enqueued into the command queue, (step a in Figure 24 on page 79). *scheduler* commands, such as the SETTIME commands originating from the data monitor applications (for example, *tis_server*), (step 8 in Figure 24 on page 79), are sent with the *TPNOREPLY* flag set, which means there will be no reply (no returned result in the result queue).

scheduler servers generate ouput to log files, Tuxedo queues, and Tuxedo servers. The updated scheduling states are enqueued to the schedule queue (Q1 in Figure 24 on page 79). Output to Tuxedo services consists of service calls to data monitor servers. *schedclient* generates output to the terminal or message window and to the sched-command queue (Q2 in Figure 24 on page 79).

Control

scheduler start up and shut down are handled by Tuxedo because *scheduler* is a Tuxedo application server. Start up upon system boot up is initiated by an operator as is manual start up and shut down of one or more of the replicated *scheduler* servers. However, Tuxedo actually handles process execution and termination. Tuxedo also monitors *scheduler* servers and provides automatic restart upon any unplanned server termination.

schedclient is always started as part of an operator request. The request can be direct by submission of the *schedclient* command within a UNIX shell environment or indirect by the operator GUI *tuxpad* (specifically by the *schedule_it* GUI).

Interfaces

The interface to the scheduling system is through the *schedclient* application, which sends commands to *scheduler*. Commands exist to initialize or re-initialize the schedule service table (Q1 in Figure 24 on page 79), add new services, delete existing services, stall services, unstall services, display the current schedule service table, and enable or disable the *scheduler* server's ability to call services. The schedule commands sent by *schedclient* are passed to the *scheduler* server via the `tpacall()` Tuxedo API function for asynchronous service calls. The string-based commands are packed into a Tuxedo STRING buffer, which is interpreted by *scheduler*. The *scheduler* server does not return any data to *schedclient*, but with the *show* command *scheduler* enqueues the service list in text form to the result queue, (step 5 in Figure 24 on page 79). *schedclient* polls the result queue waiting for the show command result, (step b in Figure 24 on page 79).

In practice, *schedclient* commands are handled by the *schedule_it* GUI, which is part of the *tuxpad* operator console, (*tuxpad* in Figure 24 on page 79).

Error States

scheduler can fail during start up if the user parameter file is non-existent or contains invalid settings. Start up errors are recorded in the local Tuxedo ULOG file of the machine hosting the failed *scheduler* server. In general, the scheduling system is designed to continue operation during system failures such as a Tuxedo queuing system error, which may only be transient in nature. Because the schedule state is stored in a reliable disk queue, failures will not result in anything more than rolling back state and retrying until the problem is fixed. The replicated fault-tolerant design of the scheduling system allows for continued successful system scheduling during *n*-1 scheduler server failures when *n* replicated servers are configured.

schedclient is relatively simple and may only fail to submit commands to the schedule command queue if the Tuxedo queuing system is unavailable or has failed. Notice of such failures is immediate and failures are reported to the user via the controlling environment, be it a command shell or the *tuxpad* GUI message window.

tuxshell

IDC Automatic Processing applications such as *DFX* and GA are not DACS servers or clients. Rather, they are child processes of the generalized processing server *tuxshell*. *tuxshell* satisfies the system requirements for support of basic, but reliable, pipeline process sequencing. Pipeline process sequencing requires application software execution and management within a transactional context. *tuxshell* performs the following functions as a transaction when called by a *TMQFORWARD* (or another *tuxshell*) (Figure 25):

1. Receive the message that was dequeued from the source queue by the *TMQFORWARD* that is upstream in the processing sequence, or receive the message from another *tuxshell* if within a compound *tuxshell* processing sequence.
2. Extract certain parameters from the message (for example time, end time, and station name for a processing interval).
3. Use these parameters to create a command line that calls an application program and contains a set of parameters/parameter files.
4. Spawn a child process by passing the command line to the operating system.
5. Update the appropriate row in the **interval** or **request** table to status *xxx*`_started` with the name of the application program replacing *xxx*.
6. Monitor the outcome of processing, and
   - if successful (as determined by the child process's return code) enqueue a message into the next queue in the processing sequence and update **interval**.*state* to `done_`*xxx*, or call another specified *tuxshell* in the case of a compound *tuxshell* processing sequence.
   - in case of failure (as determined by the child process's return code) requeue the message into the source queue, update **interval**.*state* to `retry`, and increment the retry count; or, if the retry count has been exceeded, place the message in the failed queue and update **interval**.*state* to `failed_`*xxx*.

- in case of time out (as determined by the processing time exceeding a configured value) kill the child process, requeue the message into the source queue, update **interval**.*state* to `retry`, and increment the time-out retry count; or, if the time-out retry count has been exceeded, place the message into the failed queue and update **interval**.*state* to `timeout-`*xxx*.
- go to sleep (await next service call).

The preceding list is applicable to *tuxshell* for Automatic Processing. For the Interactive Processing, database operations are absent (in other words, no **interval** table updates), and an additional reply message (success or failure) is sent to the sender (for example, *ARS*), the value of which is equal to the return code of the child.

tuxshell works in a transactional mode. *tuxshell* rolls back any changes to the queues and the **interval/request** table if some error (other than a failure of the application program) occurs. Application program failures, both orderly ones with non-zero return codes and ungraceful terminations, are handled through the retry/failed mechanism described previously. However, child processes access the database independently and not through the DACS, so they are responsible for ensuring the rollback upon abnormal termination or time out.

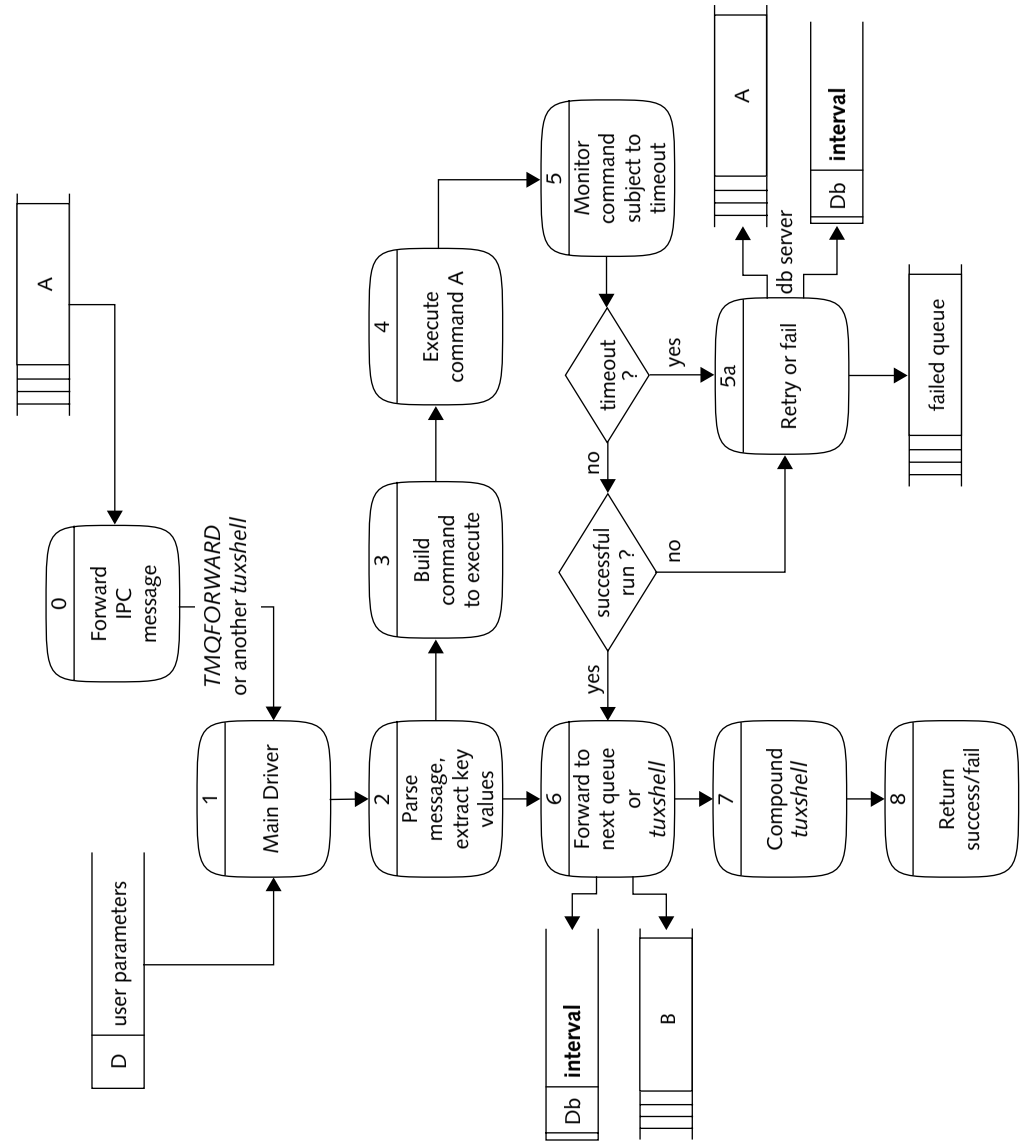


FIGURE 25. TUXSHELL DATA FLOW

Input/Processing/Output

Figure 25 on page 85 shows *tuxshell*'s data and processing flow. *tuxshell* receives input from user-defined parameter files and IPC messages through a Tuxedo service call. The Tuxedo service call originates from a *TMQFORWARD* server or another *tuxshell* (processes 0 and 1 in Figure 25 on page 85). The parameter files and IPC message specify all processing details for a given instance of the *tuxshell* server. Details include the name of the application program to be executed and managed, various keys and values used in the construction of the application program command line, database state values, processing sequencing values, and the name of the database service used for database updates. The user parameters are used to execute and manage the application program, and forward, retry or declare "failed" the application within the context of a pipeline processing sequence, all within a Tuxedo transaction.

tuxshell parses the IPC message to retrieve values to build the application program command line to be executed (process 2 in Figure 25 on page 85). The IPC message is string-based and contains name/value pairs in *libpar(3)* fashion. The values extracted from the message are limited to the name key values that are user-defined. Typically, a station or network *name*, *time* and *endtime* will be included in the name key values. This is true in general because *tuxshell* manages the processing of an application server that operates on an interval of time. The elements of the command line are user defined and allow for the substitution of the parsed values (process 3 in Figure 25 on page 85). The completed command line is executed (process 4 in Figure 25 on page 85), and *tuxshell* then initiates monitoring of the child process. Monitoring of the application server includes capturing the exit code of the process if it terminates in a normal manner, killing the process if a time-out condition arises, and detecting an abnormal termination following various UNIX *signal* exceptions (process 5 in Figure 25 on page 85).

A normal application program run terminates with an exit code indicating success or failure, subject to user-specified exit values. A successful match of the exit code results in an attempt to forward the processing interval to the next Tuxedo queue or to the next *tuxshell* depending on user parameters (process 6 in Figure 25 on page 85). Successful forwarding is always coupled with a database update via a

service call to the database server, *dbserver*. Forwarding failures in the form of a database service request failure, Tuxedo enqueue failure, or failure of the next *tuxshell* service request, result in a rollback of processing. The rollback is Tuxedo queue based wherein the transaction opened by the calling *TMQFORWARD* is undone, and the IPC message is returned to the source queue. In the case of *tuxshell* compound processing, where one *tuxshell* is called by another *tuxshell* (process 7 in Figure 25 on page 85), the service requests are unwound by failure returns, and the original transaction from the originating *TMQFORWARD* is rolled back.

Illegal exit codes, application server timeout, or abnormal process terminations are handled by *tuxshell* in a similar manner. Basically, processing intervals are either retried or declared failed subject to a user-specified maximum number of retries (process 5a in Figure 25 on page 85). Retry processing results in requeuing the interval into the source queue. Error processing results in enqueueing the interval into the user-specified failure queue. *tuxshell* queuing operations are always coupled with database updates via service calls to *dbserver*, and both operations are part of one transaction. Failure of either operation results in a transaction rollback as described above.

tuxshell generates output to log files, the database (via *dbserver*), and Tuxedo queues. Output to the database includes updates to the **interval** or **request** tables. Database updates are coupled with enqueues as described above.

Within the context of the Interactive Processing, *tuxshell* supports all previously described processing with one exception and one addition. An IPC request from an Interactive Processing client (for example, *ARS*), results in *tuxshell* returning the exit value directly back to the calling client via an IPC message. In addition, an IPC event is sent to the DACS client, *dman*. This IPC event is consistent with IPC messaging within the interactive where any message send or receive is accompanied by a broadcast to *dman* notifying this client of each message operation within the interactive session. The acknowledgement IPC message and event are not coupled with any database updates via *dbserver* requests. Essentially, the application program that is run on behalf of the interactive client is run on-the-fly and is of inter-

est only to the analyst who owns the interactive session. The pipeline operator is not interested in monitoring these intervals (for example, via the *WorkFlow* display).

Control

Tuxedo controls the start up and shut down of *tuxshell*, because *tuxshell* is a Tuxedo application server. However, *tuxshell* can also be manually shut down and booted by the operator. Tuxedo actually handles all process execution and termination. Tuxedo also monitors *tuxshell* servers and provides automatic restart upon any unplanned server termination.

Interfaces

Operators use the Tuxedo command line administration utilities directly or indirectly by *tuxpad* to manually boot and shut down *tuxshell*.

Error States

tuxshell can fail during start up if the user parameter file is non-existent or contains invalid settings. Start up errors are recorded in the local Tuxedo ULOG file of the machine hosting the failed *tuxshell* server. *tuxshell* error handling of the application server child process is fairly extensive and is described in "Input/Processing/Output" on page 78.

tuxshell servers benefit from server replication, wherein a given *tuxshell* instance can be replicated across more than one machine. In this scenario, recovery from any server or machine failure is seamless because the replicated *tuxshell* server takes over processing. Tuxedo recovers the program crash by automatically restarting the server.

dbserver, interval_router, and recycler_server

dbserver

dbserver provides an interface between the ORACLE database and DACS servers. All instances of *tuxshell* within the context of Automatic Processing operate on the **interval** or **request** table in the database through *dbserver*. Any number of *tuxshell* servers send database update statements to one of several replicated *dbservers*. In turn, *dbserver* submits the database update to the ORACLE database server (Figure 26). This setup has the advantage that fewer database connections are required. Conservation of database connections and/or concurrent database connections is at least an implicit system requirement, and as such, inclusion of *dbserver* within the pipeline processing scheme of DACS was an important design decision.

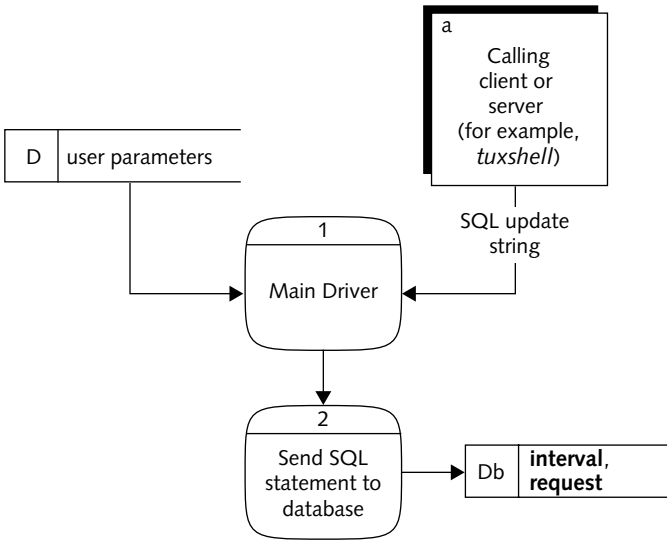


FIGURE 26. DBSERVER DATA FLOW

interval_router

The routing of messages to particular instances of a server for different data sources is supported by *interval_router* (process 5 in Figure 14 on page 50). Message routing is manifest in message enqueues into a set of defined queues. Each message route is a function of the message data where the user-defined parameters map data values to a particular destination queue name. Message routing can be used to ensure that detection processing of data from a particular station is directed to a specified queue. The DACS can be configured to process messages from specific queues on specific machines (for example, a machine that physically holds the corresponding diskloop on a local disk). *interval_router* can also be used to implement data dependent routing (for example, to make a distinction between seismic and infrasonic stations).

recycler_server

Under certain system error conditions queue messages may be diverted to the error queue. For example, replicated servers that advertise a service may become unavailable if an operator inadvertently shuts down all servers that advertise the service. A *TMQFORWARD* could subsequently try to send the message to the now unavailable service. In case of such a failure the message ends up in the error queue, perhaps after failed attempts by the *TMQFORWARD*. An operator could attempt to manual recover this message (recover the processing interval). However, *recycler_server* automatically handles retries in this failure scenario. *recycler_server* regularly checks the error queue and recycles any messages found in the error queue by placing the messages back in their original queue (processes 11 and 12 in Figure 14 on page 50).

The error queue is distinct from the failed queue that collects messages from repeated application processing failures. Reprocessing of failed intervals is handled under operator control via the workflow monitoring utility, *WorkFlow*. Application failures and subsequent reprocessing is normally part of operator's investigation into the reason for the failure. System errors which are often transient in nature are ideally automatically reprocessed. The design of *recycler_server* is influenced by the DACS system-wide requirement to provide fault tolerance.

Input/Processing/Output

dbserver

dbserver receives input from user parameters and *tuxshell* application servers. The user parameters define the ORACLE database account to which *dbserver* connects and forwards database statements. *tuxshell* servers send *dbserver* the database update messages through an IPC message string. The IPC input message consists of a fully resolved SQL statement that is simply submitted to the ORACLE database server via a standard *libgdi* call. *dbserver* further uses a *libgdi* call to commit the database submission assuming a successful database update. *dbserver* returns a *success* or *failure* service call return value to the calling *tuxshell* depending on the status of the database operation. *dbserver* logs all database statements and progress to the user-defined log file.

interval_router

interval_router receives input from user parameters and data monitor application servers. The user parameters define the mapping between interval name (same as station or sensor name) and the target Tuxedo queue name as well as the name of the qspace to which the messages will be routed. A data monitor server such as *tis_server* can optionally rely upon *interval_router* for enqueuing new intervals into Tuxedo queues. A *tis_server* sends *interval_router* the interval IPC message, and *interval_router* performs the enqueue operation as a function of the interval name. The interval name is extracted from the interval message. The name is extracted by the Tuxedo FML32 library, which provides an API interface for reading from and writing to Tuxedo IPC messages. The interval message source and destination fields are set by *interval_router* to conform with the DACS interval message format standard (see *libipc* below for details). *interval_router* then attempts to map the interval name to the target queue as defined by the user parameters. *interval_router* returns a success or failure service call return value to the calling *tis_server* depending on the status of the mapping and/or enqueue operation. *interval_router* logs all routing progress to the user-defined log file.

recycler_server

recycler_server receives input from user parameters and a *TMQFORWARD* server. The user parameters define the name of the qspace to which messages will be recycled. *TMQFORWARD* monitors the error queue and sends any available messages in that queue to *recycler_server*. *recycler_server* extracts the source service name (which is the queue name) from the interval message. Like *interval_router*, the source service name is extracted by the Tuxedo FML32 library. *recycler_server* resets the failure count and timeout count to zero by updating the corresponding fields in the interval message. This is done because the recycled message is intended for retry as if it were a new interval with no previous failed attempts. *recycler_server* then attempts to enqueue the revised interval message to the originating queue. *recycler_server* returns a success or failure service call return value to the calling *TMQFORWARD* depending on the status of the enqueue operation. *recycler_server* logs all routing progress to the user-defined log file.

Control

Tuxedo controls the start up and shut down of *dbserver*, *interval_router*, and *recycler_server*, because *dbserver*, *interval_router*, and *recycler_server* are Tuxedo application servers. However, *dbserver*, *interval_router*, and *recycler_server* can also be manually shut down and booted by the operator. Tuxedo controls all actual process executions and terminations. Tuxedo also monitors the servers and provides automatic restart upon any unplanned server termination.

Interfaces

Operators can assist in the control of *dbserver*, *interval_router*, and *recycler_server* by using the Tuxedo command line administration utilities directly or indirectly via *tuxpad*.

Error States

dbserver, *interval_router*, and *recycler_server* can fail during start up if the user parameter file is non-existent or contains invalid settings. Start up errors are recorded in the local Tuxedo ULOG file of the machine hosting the failed server. Service failure, including database submit failure in the case of *dbserver* or enqueue failures in the case of *interval_router* and *recycler_server,* result in failure return codes to the calling servers as described above. In each case, the calling server handles these service failures.

These application servers benefit from server replication wherein a given server instance can be replicated across more than one machine. In this scenario, recovery from any server or machine failure is seamless because the replicated server takes over processing. Tuxedo recovers the failure of a *dbserver*, *interval_router*, or *recycler_server* due to a program crash by automatically restarting the server.

Database connection management is included in *dbserver*. An application server such as *dbserver* runs for long periods of time between reboots, and so on. *dbserver*'s runtime duration might exceed that of the ORACLE database server. In general these design goals are satisfied by management of the ORACLE database connection, such that a temporary disconnect or failure can be retried after a wait period.

WorkFlow, SendMessage, and ProcessInterval

WorkFlow provides a graphical representation of time interval information in the system database (the **interval** and **request** tables). *WorkFlow* satisfies the system requirement to provide a GUI-based operator console for the purpose of monitoring the progress of all automated processing pipelines in real or near real time. The current state of all automated processing pipelines is recorded in the *state* column of each record in the **interval** and in the *status* column of the **request** database table.

WorkFlow visualizes the Automatic Processing pipeline and progress of analyst review by displaying rows or timelines organized by pipeline type or class (for example, TI/S - time interval by station) and processing name or station (for exam-

ple, ARCES - seismic station) (Figure 27). Each horizontal timeline row is composed of contiguous time interval columns or bricks. The *WorkFlow* brick is colored according to the interval state where the mapping between state and color is user defined. The timeline axis is horizontal with the current time (GMT) on the right side. All interval bricks shift to the left as time passes, and newly created intervals occupy the space on the right. The *WorkFlow* design enables convenient scaling of the amount of interval information displayed on screen. The horizontal pixel size of each time block is reduced or enlarged depending on the number of intervals displayed. The GUI-based controls enable the operator to adjust the history or number of intervals hours and duration, which is essentially the horizontal size of each *WorkFlow* brick.

A requirement also exists to enable the operator to reprocess any interval via GUI control. Intervals eligible for reprocessing are defined via user parameters and are typically limited to intervals with state(s) that define a terminal condition such as *failed*, *error*, or even *done/success*. *SendMessage* enables interval reprocessing by translating database interval information into a Tuxedo queue-based message and then routing the message to a Tuxedo queue to initiate pipeline processing for the desired interval. *ProcessInterval* is a shell script that facilitates linking *WorkFlow* and *SendMessage*.

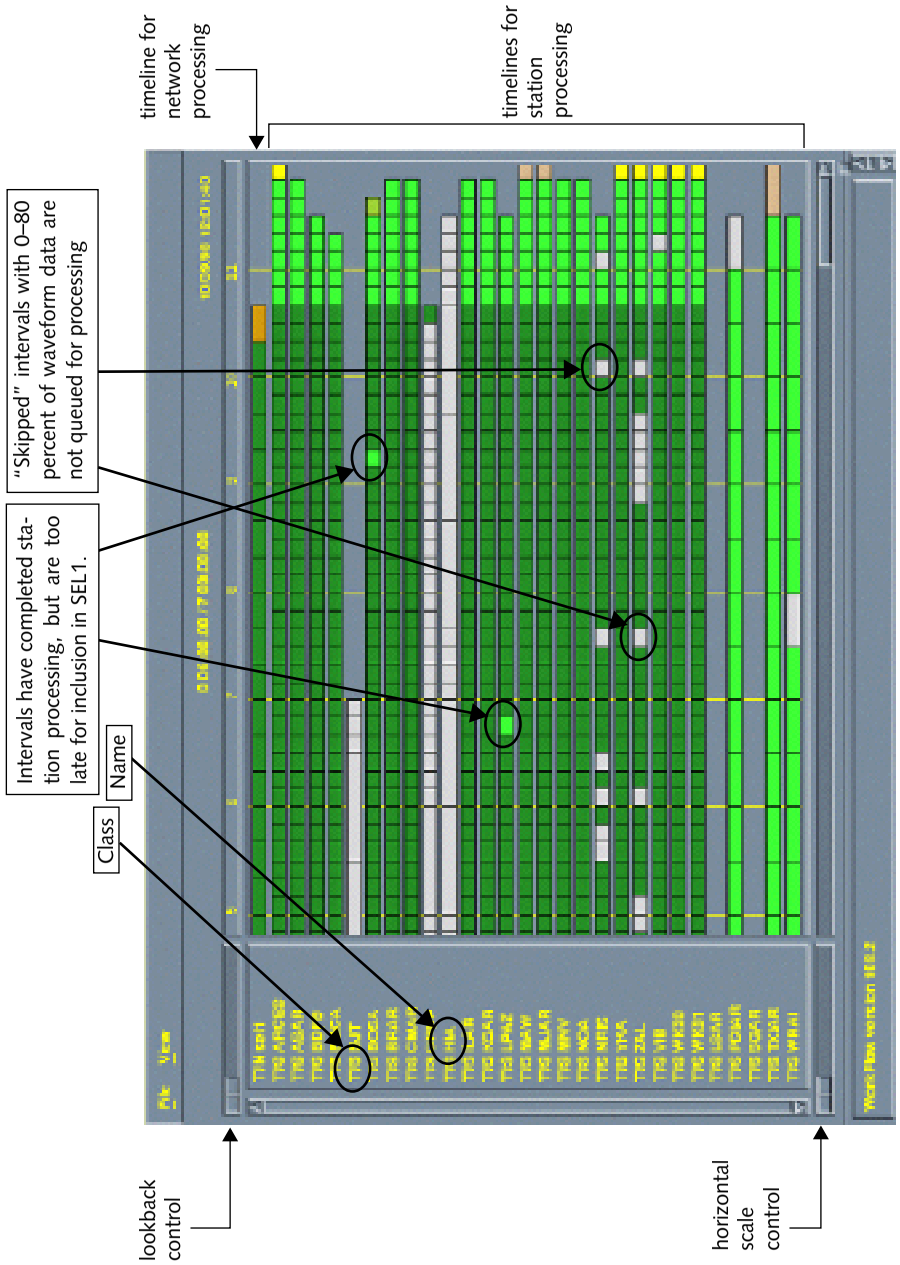

timeline for network processing
timelines for station processing
"Skipped" intervals with 0–80 percent of waveform data are not queued for processing
Intervals have completed station processing, but are too late for inclusion in SEL1.
Name
Class
lookback control
horizontal scale control

FIGURE 27. MONITORING UTILITY WORKFLOW

Input/Processing/Output

WorkFlow receives input from three sources including user parameters, the database, and the user via manipulations and selections of the GUI. The user parameters specify database account values, query options, and definitions for all classes and names of time intervals that *WorkFlow* will monitor.

WorkFlow maintains an internal table of all time intervals. The size of the table can be significant because *WorkFlow* is required to display tens of thousands of bricks, which can span a number of timelines (easily 100) and hundreds of intervals on each timeline. Access to the table for interval updates must be fast enough to avoid interactive response delays in the GUI. To meet these requirements, *WorkFlow* is designed around a hash table, which achieves O(1)[11] based access for nearly instantaneous, specific interval recall. The hash table is shown as an internal data structure (M1 in Figure 28). The hash table is built during *WorkFlow* initialization where all time intervals, subject to a user-specified time lookback, are retrieved from the database. The construction of the hash table can be expensive but the initialization or start-up delay is still bounded by the database *select* on the **interval** or **request** table.

11. O- notation or order-notation is used to quantify the speed characteristics of an algorithm. For example, a binary search tree would be $O(\log_2)$ or on order log base two search time. O(1) implies direct lookup which is optimal.

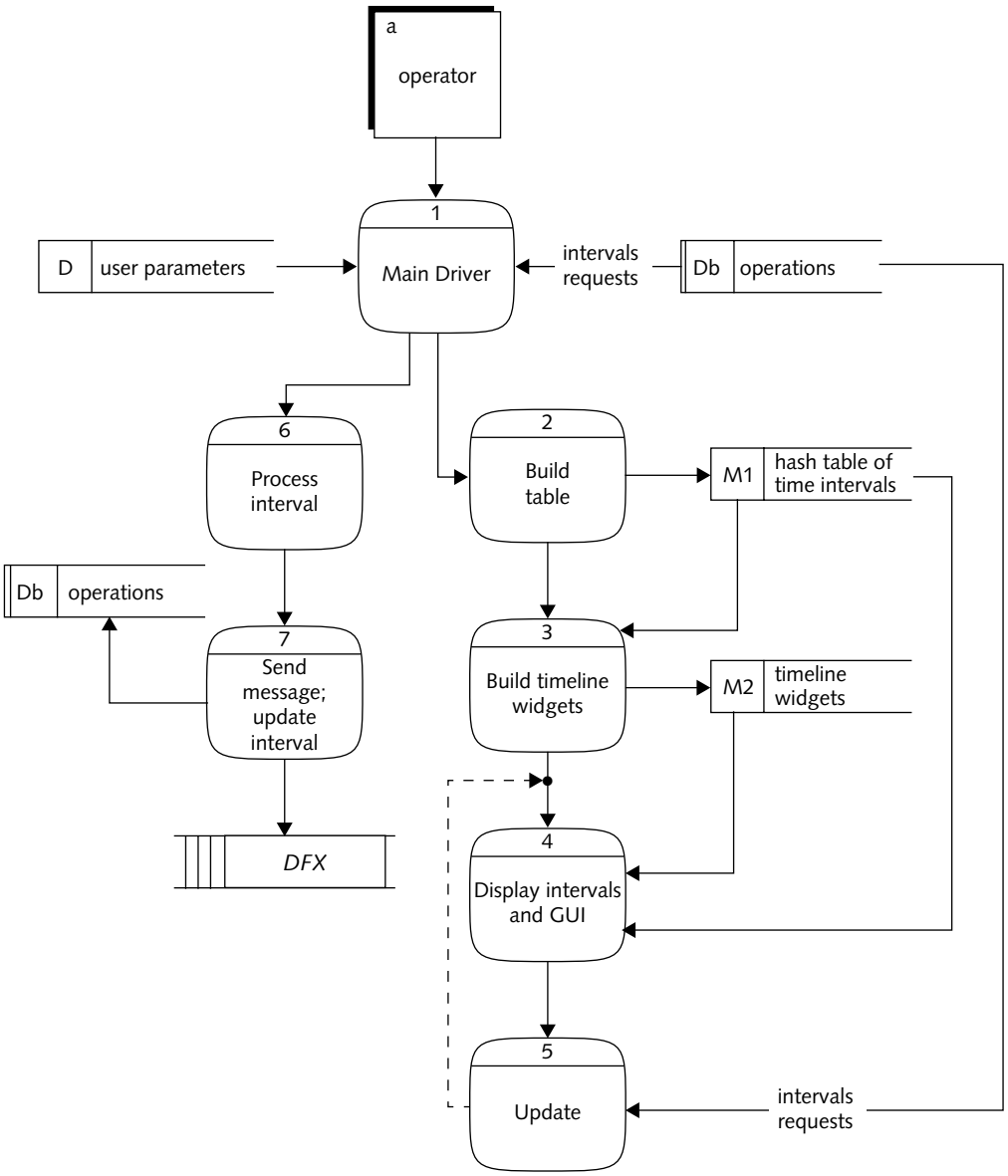


FIGURE 28. WORKFLOW DATA FLOW

The hash table is updated during every *WorkFlow* update cycle. The *WorkFlow* update cycle consists of interval database queries where the *select* is confined to all intervals of interest that have *moddate* values within the previous 5 minutes. Retrieved rows include time intervals that have changed state as well as new time intervals. This current interval information is used to update or add to the hash table.

Input to *WorkFlow* via the GUI consists of pointer selections to vertically scroll through the list of timelines, horizontally scroll across all time intervals, scale the interval history and duration, retrieve interval class, name, times, state, and interval identifier from a specific brick, and reprocess a specific brick (if enabled). Additionally, GUI input is accessible via pull-down menus that enable *WorkFlow* re-initialization, update on demand, display of exception (failed) intervals, and display of a color-based legend for color/state mappings. All GUI input results in exercising various control and interface functions that are described in the following sections.

WorkFlow output is primarily defined by the GUI display and is in part under user control as described above. The update cycle is automatic (and manual via a menu selection), and results in an updated visualization of the hash table. *WorkFlow* diagnostics are sent to the GUI message window at the bottom of the *WorkFlow* display. *WorkFlow* error messages (particularly of the fatal variety) are sent to the controlling terminal when the GUI message window is not yet displayed.[12]

ProcessInterval and *SendMessage* are driven by *WorkFlow*, and as such their input is provided by *WorkFlow*. Both the *ProcessInterval* C-Shell script and the *SendMessage* program can be run stand-alone, although in practice *SendMessage* is the only candidate for usage outside of *WorkFlow*. *WorkFlow* is typically configured to run *ProcessInterval* upon user selection of interval reprocessing. In turn, the script builds a *SendMessage* command line and then runs the command. The *SendMessage* command line includes all interval values including *class*, *name*, *time*, *endtime*, *state*, and interval identifier. *SendMessage* attempts to enqueue the interval information into a Tuxedo queue. *SendMessage* is a Tuxedo client application that uses the Tuxedo `tpenqueue()` API call to send to the Tuxedo queue (processes 6 and 7 in Fig-

12. The *WorkFlow* GUI is not displayed if a fatal error occurs during startup.

ure 28). *SendMessage* output is sent to the controlling terminal, which is *WorkFlow* in this case. *WorkFlow* redirects *SendMessage* output to the *WorkFlow* message window, which reports the results of command.

Control

WorkFlow is an interactive client application and is started and shut down by system operators. *WorkFlow* is primarily designed for monitoring and is therefore primarily a read-only tool. However, interval reprocessing and other possible write-based operations are available. As such, *WorkFlow* is typically started via shell scripts that limit access to read only for public monitoring of the automated pipeline processing system and allows full access for the pipeline operators. *WorkFlow* start shell scripts also exist for convenient monitoring of the **request** table.

The *ProcessInterval* shell script is run by *WorkFlow* as described in the previous section. The *SendMessage* application is run by *ProcessInterval* in the *WorkFlow* context (also described above). The *SendMessage* client can be run stand-alone and usage is similar to any standard command line application except that as a Tuxedo client application, *SendMessage* must be run on an active Tuxedo host.

Interfaces

The *WorkFlow* GUI is designed around the expectation of a relatively high performance graphical subsystem that is accessible through a high-level programming interface that likely includes an abstract class-based GUI toolkit. The GUI toolkit should enable extension so that new GUI components can be created if required for unique feature requirements, speed, or implementation convenience.

WorkFlow is currently implemented using the X11 Window System using the Xlib, Xt, and Motif toolkits and libraries. The GUI design and layout relies upon widgets for a graphical canvas (main form) upon which pull-down menus, scroll bars, scale bars, a message window, and the main form windows for brick and class/name display can be constructed in one GUI widget hierarchy. The displayed timelines are handled via a custom timeline widget that controls display and management of each brick on the timeline.

The *ProcessInterval* and *SendMessage* interfaces and interaction between each other and *WorkFlow* are described in the previous sections.

Error States

WorkFlow errors and failures can occur at program initialization or during program execution. The most typical error state is invalid or incomplete user parameters. User parameters define the time interval classes, state to color mappings, interval reprocessing commands, as well as database account and query information. Incorrect database parameters usually result in *WorkFlow* termination. Incomplete color/state specification can result in program termination or unexpected and confusing color mappings. Insufficient color map availability is a common error state whereby *WorkFlow* will not even start. *WorkFlow* provides/produces relevant error messages to direct the user to a solution.

Runtime *WorkFlow* errors are most typically associated with a database server failure where, for example, the server may go away for a period of time. *WorkFlow* has been designed to survive a database server outage via recurring attempts to reconnect to the database server and resume normal continuous monitoring.

ProcessInterval errors are probably due to invalid user parameters, which should become apparent via error messages provided to the *WorkFlow* GUI message window. *SendMessage* errors should only occur if the Tuxedo queuing is not available or the Tuxedo qspace is full, both of which would be indicated in the GUI message window.

libipc, dman, and birdie

libipc and *dman* satisfy requirements for DACS support of distributed asynchronous messaging between Interactive Tools, management of an interactive session through the monitoring of messages and Interactive Tools within the session, and execution of Interactive Tools on demand. All Interactive Tools (for example, *ARS*, *dman*, and *XfkDisplay*) link to and use *libipc* for message passing and session management.

libipc consists of an Application Programming Interface (API) or library of routines, which enable reliable distributed asynchronous messaging and message and client monitoring within an interactive session. *dman* is a GUI-based interactive client with special bindings to the *libipc* library to enable session monitoring and management. *birdie* is a command-line-based application, which is primarily intended as a test driver to exercise the *libipc* API. *birdie* permits arbitrary access to all session-level functions (for example, delete a message in a queue), and as such can be used by operators either directly or via embedding in scripts to perform certain manipulations on queries.

Figure 15 on page 53 shows the data flow of DACS CSCs for Interactive Processing. The data flow among the various processes and DACS is described in "Data Flow Model" on page 48. The messages exchanged between the Interactive Tools (all *libipc* messages) pass through Tuxedo disk queues. Storing messages within a disk based Tuxedo queue ensures that the messaging is asynchronous, because the message send and receive are part of separate queuing operations and transactions. For example, under analyst control (a in Figure 15 on page 53), a message sent from *ARS* (process 2 in Figure 15 on page 53) intended for *XfkDisplay* is enqueued by *libipc* into the *XfkDisplay* queue. Asynchronous messaging allows for the possibility that *XfkDisplay* may not be currently running in the analyst's interactive session. *libipc* uses Tuxedo-based events (memory-based broadcast messages) to signal *dman* for each message send or receive within the interactive session (processes 3 and 1 in Figure 15 on page 53). The Tuxedo server, TMUSREVT (not shown in Figure 15), processes all user events for Tuxedo clients and servers. The event processing includes notification and delivery of a posted event (for example, from *ARS*) to all clients or servers that subscribe to the event or event type (for example, *dman*). *dman* tracks the processing status of all clients within the analyst's interactive session via *libipc*. *dman* executes *XfkDisplay* on demand if it is not already running (process 4 in Figure 15 on page 53). *dman* uses the processing status for each client to visually present to the analyst the status of that client. *dman* monitors all message traffic within the interactive session via the *libipc* events described above and can therefore keeps track and visually display the consump-

tion and creation of messages. In addition, *dman* can query the number of queued messages for any session client/queue, which is required at session start up to determine the absolute number of pending messages in each queue.

The relationship between *libipc* and DACS for Automatic Processing is limited and nonexistent for the purposes of the *dman* client. However, *libipc* defines the structure of the IPC messages that are used within Automatic Processing and Interactive Processing as well as between these subsystems. *ARS* relies upon Automatic Processing for interactive recall processing such as *DFX*-based Beam-on-the-Fly (BOTF) processing. Recall processing depends upon a standard *libipc*-based message send by *ARS* to the BOTF queue, which is configured within the interactive session queuing system (processes 2 and 5 in Figure 15 on page 53). The *TMQFORWARD*/*tuxshell* configuration for managing Automatic Processing applications works in a similar but not identical manner to DACS for Interactive Processing (processes 5–7 in Figure 15 on page 53). *TMQFORWARD* calls a *tuxshell* server within a transaction, but the processing application status, success or fail, is sent back to the calling client via a *libipc* message (process 6 in Figure 15 on page 53). However, the message is not entirely *libipc* compliant in that *tuxshell* does not send an IPC broadcast to the interactive session *dman* client.[13] Finally, *tuxshell* does not attempt an **interval**.*state* update in the databases because this processing is on-the-fly and is not represented as an interval in the database.

The structure of messages within DACS for both Interactive Processing and Automatic Processing is defined by *libipc* and is described in detail in Table 3. The first column of Table 4 lists the message attribute name, the middle column maps any relationship to the database **interval/request** table, and the third column defines the attribute and explains how it is used within DACS for both Interactive and Automatic Processing.

The design decision to base *libipc* messaging on Tuxedo disk queuing was influenced by several criteria including convenience, history, and implementation time constraints. The implementation was convenient because messages within DACS for Automatic Processing are based upon Tuxedo queues, and Interactive Process-

13. In practice, the lack of the IPC event message does not cause any problems.

ing and Automatic Processing can exchange messages. As a result a unified messaging model across the two systems, which exchange messages, was implemented. In practice, Interactive Processing and Automatic Processing run in separate DACS applications, and as such the messaging does not cross between the systems. However, this configuration was not anticipated and therefore was not part of the design decision. Earlier DACS implementations had also successfully used the unified model. The *TMQFOWARD/tuxshell* scheme is re-used within the Interactive Processing configuration, and as such some leveraging is realized even though the systems run in separate applications. It would be possible to re-implement DACS for Interactive Processing based upon a messaging infrastructure separate from Tuxedo. Such an implementation would likely have to include a gateway or bridge process to pass messages from Interactive Processing to the Tuxedo-based DACS for Automatic Processing.

TABLE 3: DACS/LIBIPC INTERVAL MESSAGE DEFINITION

Field Name	Database Interval	Description
1 *MSGID*	**interval**.*intvlid*	Each Tuxedo queue message can have a unique identifier assigned at the application level (not assigned by Tuxedo, which assigns its own identifier to each queue message for internal purposes). This unique identifier is known as the queue correlation ID (CORRID), and this value can be used for query access to the queue message (for example, to delete or read the message out of normal First In First Out (FIFO) queue order). DACS sets MSGID (CORRID) to the value of **interval**.*intvlid,* thereby linking the queue interval message to the database interval record.
2 *MSGSRC*	N/A	This field stores the source qspace name and queue. The source is sometimes referred to as the sender, as in the sender that initiated the message send.
3 *MSGDEST*	N/A	This field stores the destination qspace name and queue. The destination is sometimes referred to as the receiver, as in the recipient that receives the delivered message.

TABLE 3: DACS/LIBIPC INTERVAL MESSAGE DEFINITION (CONTINUED)

Field Name	Database Interval	Description
4 *MSGCLASS*	N/A	This field stores the class of the message, which is generally used to distinguish queue messages between the Automatic and Interactive Processing DACS applications.
5 *MSGDATA*	**interval**.*time/endtime/name/class/state/intvlid*, **request**.*sta/array/chan/class/start_time/end_time/reqid*	For messages sent to or within Automatic Processing, *MSGDATA* stores **interval** or **request** information. These messages originate from either DACS data monitors or an Interactive Tool such as *ARS*. The *tuxshell* server extracts this message value as a string and then parses time, class, and name values used to construct the automated processing application command line. For messages returned to an Interactive Tool from *tuxshell*, *MSGDATA* stores a success or fail code/string that represents the status of the automated processing application. For messages within Interactive Processing, *MSGDATA* stores string-based IPC messages relevant to the sender and receiver Interactive Tools. These IPC messages may include algorithm parameters, database account and table names, file path names, scheme code, and so on.
6 *MSGDATA2*	N/A	This field stores interval priority assigned by a DACS data monitor. DACS queuing optionally supports out of order dequeuing (for example, via *TMQFORWARD*) based upon interval priority. The data monitor server, *tis_server*, can enqueue new intervals such that more recent or current data are processing before older or late arriving data.
7 *MSGDATA3*	N/A	This field stores application processing timeout failcounts, which are managed by *tuxshell*.
8 *MSGDATA4*	N/A	This field is reserved for future use.
9 *MSGDATA5*	N/A	This field is reserved for future use.
10 *FAILCOUNT*	N/A	This field stores application processing failcounts, which are managed by *tuxshell*.

Input/Processing/Output

Input and outputs within *libipc* are largely based upon the details or call semantics of the API. Important details related to the *libipc* API are listed in Table 4. The first column in the table lists the API call name. The second column describes the call. The third column indicates if the call is used by any of the DACS CSCs for Automatic Processing. In general, the DACS CSCs for Automatic Processing do not rely upon *libipc* for their messaging, and their usage is limited to fairly trivial convenience functions. The fourth column indicates which Interactive Processing DACS clients use the API call. The final column briefly notes the API call's usage of queuing, events, and Tuxedo Management Information Base (MIB) calls. The Tuxedo MIB API provides for querying and changing the distributed application.

dman input and output, beyond that already described and related to *libipc*, is described in the *Interactive Analysis Subsystem Software User Manual* [IDC6.5.1] and the *dman* man page, *dman(1)*.

birdie is a command-line-driven program, and its inputs and outputs are described in the *birdie* man page, *birdie(1)*.

TABLE 4: LIBIPC API

	API function	Description	DACS Automatic Processing Usage	DACS Interactive Processing Usage[1]	IPC Queue/ IPC Event/ Tuxedo MIB Usage
1	*ipc_attach()*	Attaches calling client to the IPC session defined by the `QSPACE` environment variable and the *group* and *name* arguments. Returns a pointer to an *ipcConn* object, which provides access to the IPC session for this client.	N	all	Uses a message enqueue to test the specified default IPC queue.
2	*ipc_detach()*	Detaches calling client from the IPC session pointed to by the *ipcConn* object argument.	N	all	N/A
3	*ipc_send()*	Sends a message to the specified message queue within the IPC session pointed to by the *ipcConn* object argument.	N	all except *dman*	Uses a message enqueue and an event broadcast to the session's *dman* client.
4	*ipc_receive()*	Retrieves the next message in the specified queue within the IPC session pointed to by the *ipcConn* object argument.	N	all except *dman*	Uses a message dequeue and an event broadcast to the session's *dman* client.
5	*ipc_check()*	Returns boolean true if a new message has arrived to the default queue since the last *ipc_receive()* call. The default queue is the queue name provided during the *ipc_attach()* call and is defined in the *ipcConn* object. This function always returns boolean true due to an implementation change to *libipc*.[2]	N	all except *dman*	N/A

TABLE 4: LIBIPC API (CONTINUED)

API function	Description	DACS Automatic Processing Usage	DACS Interactive Processing Usage[1]	IPC Queue/ IPC Event/ Tuxedo MIB Usage
6 *ipc_pending()*	Retrieves the number of messages queued for the list of queue names specified.	N	*dman* and *birdie* only	Uses Tuxedo MIB calls to retrieve the number of messages.
7 *ipc_purge()*	Deletes first or all messages from the specified queue.	N	*dman* and *birdie* only	Uses message dequeue(s) to purge queue message(s).
8 *ipc_client_status()*	Retrieves the processing status for each client defined in the list of specified clients.	N	*dman* and *birdie* only	Uses Tuxedo MIB calls to determine the client processing status.
9 *ipc_add_xcallback()*	Registers a client callback function, which is invoked periodically for the purposes of polling an IPC queue. Presumably the callback function will use *ipc_receive()* to retrieve IPC messages. The frequency of the callbacks is currently fixed at two times per second.[3]	N	all except *dman*	N/A
10 *ipc_remove_xcallback()*	Removes the client callback function from the clients *libXt* Xtoolkit event loop.	N	all except *dman*	N/A
11 *ipc_get_error()*	Retrieves error status following all *libipc* calls and detailed error information for any error conditions.	N	all	N/A

TABLE 4: LIBIPC API (CONTINUED)

API function	Description	DACS Automatic Processing Usage	DACS Interactive Processing Usage[1]	IPC Queue/ IPC Event/ Tuxedo MIB Usage
12 *ipc_get_group*()	Convenience function that extracts the IPC group name given the specified IPC queue name (IPC address).	N	all except *dman*	N/A
13 *ipc_get_name*()	Convenience function that extracts the IPC name given the specified IPC queue name (IPC address).	Y	all except *dman*	N/A
14 *ipc_make_address*()	Returns the IPC address (IPC queue name) based upon the specified IPC group and name.	Y	all except *dman*	N/A

1. *libipc*-based clients that are relevant to the DACS for Interactive Processing include *dman*, *birdie*, *ARS*, *XfkDisplay*, *Map*, *PolariPlot*, *SpectraPlot*, *IADR*, and *AEQ*.

2. The *ipc_check()* call was intended to enable a check for pending queue messages without an actual message read/dequeue. Problems with the Tuxedo unsolicited message handling feature required an implementation change wherein polling is carried out via explicit calls to *ipc_receive()*. The implementation change included making *ipc_check()* always return true, which in effect forces an *ipc_receive()* call for every client-based attempt to check for any new messages.

3. The callbacks are added to the clients *libXt*-based Xtoolkit event loop in the form of a timer-based event via the *XtAppAddTimeOut() libXt* call.

Control

libipc is a library and is therefore not explicitly started or stopped but is instead embedded or linked into client applications.

dman is started by the analyst either manually, via the desktop GUI environment such as the CDE, or via the *analyst_log* application. The *dman* GUI is controlled by the analyst. *dman* is typically stopped via a script, which is bound to a CDE button, or *dman* can be terminated by selecting the *dman* GUI's exit menu option.

birdie is started, controlled, and stopped by an operator or via a script that embeds *birdie* commands within it.

Interfaces

The exchange of data and control among *libipc* and its clients, including *dman*, has been described in the sections "libipc, dman, and birdie" on page 100, "Input/Processing/Output" on page 105, and "Control" above.

birdie is basically a driver for *libipc,* and it exchanges data with *libipc* and other session clients via the *libipc* API. The operator provides command line input, which is interpreted by *birdie* and included within the *libipc* API calls.

Error States

The *libipc* implementation tests for many error conditions. Example errors include non-existent QSPACE environment variables, bad queue names, and attempts to send or receive messages when not attached to an interactive session. The errors are returned back to the calling client via API return codes. Error detection and detailed error codes and messages are accessible via the *ipc_get_error()* call (see Table 4 on page 106).

dman can encounter many error conditions. An example error includes a non-existent *agent* parameter specification, which prevents *dman* from running because it does not have a session to which it can connect. A non-existent `QMCONFIG` environment variable will similarly result in an immediate failure because this variable is required for message polling. One and only one *dman* per session is permitted, and

dman defends against this by exiting with a failure message indicating that the session already has an active *dman* if one exists. There are many other types of error conditions that *dman* attempts to guard against and warn the analyst. The *dman* GUI includes a message window, which conveniently presents warning messages and other diagnostics to the analyst.

birdie directs error messages to the standard error stream, which is consistent with most command-line-driven applications. *birdie* error conditions are all of the *libipc* error conditions because *birdie* is intended to exercise all *libipc* API calls.

tuxpad, operate_admin, schedule_it, and msg_window

tuxpad provides a GUI-based operator console to simplify operation of the DACS. *tuxpad* satisfies the requirement to provide a convenient centralized operator console that can be used by the operator to control all aspects of the running distributed application. *tuxpad* consists of five applications; four of them are manifested in interactive GUIs that are all accessible via the main *tuxpad* GUI. The five applications are: *tuxpad*, *operate_admin*, *schedule_it*, *qinfo*, and *msg_window*. The *schedule_it* and *qinfo* applications can optionally be run stand-alone, whereas *operate_admin* and *msg_window* are integral to *tuxpad*. All applications are designed to provide an intuitive front end to the underlying Tuxedo administrative commands (for example, *tmadmin*) and the DACS control clients (for example, *schedclient*). These front ends generate Tuxedo and the DACS client commands that are run. Their output is parsed for results that are then presented to the operator via the GUI. These primary design objectives necessitated a scripting language including flexible text parsing, support for dynamic memory and variable length lists, convenient process execution and management, and a high-level GUI toolkit. Perl/Tk, the Perl scripting language with integrated bindings to the Tk GUI toolkit, met all the requirements and is used for implementation for all five of the *tuxpad* scripts.

tuxpad drives the Tuxedo command line based administration tools: *tmadmin*, *tmboot*, and *tmunloadcf* (Figure 29). *tuxpad* also provides one button access to the *qinfo*, *schedule_it*, and *msg_window* GUIs. *tuxpad* displays all configured machines,

process groups, servers, and service names contained in the distributed Tuxedo DACS application in the main *tuxpad* window. Mapping between logical machine names and actual machine names, process group names and numbers, server names and server identifiers, and server names and service names are also displayed in the main *tuxpad* window. The mappings are interpreted following a parsing of the complete Tuxedo UBB configuration, which is generated upon execution of the *tmunloadcf* command. The mapping and current state of the machines, groups, and servers is kept current via parsing the output from the *tmadmin* command on a recurring and on-demand basis. *tuxpad* is also aware of the Tuxedo DACS notion of the backup or replicated server and is able to organize server display to conveniently present the status of both primary and backup servers. Machine, group, and server booting and shut down are handled by *tuxpad* executions of the `tmboot` and `tmshutdown` commands.

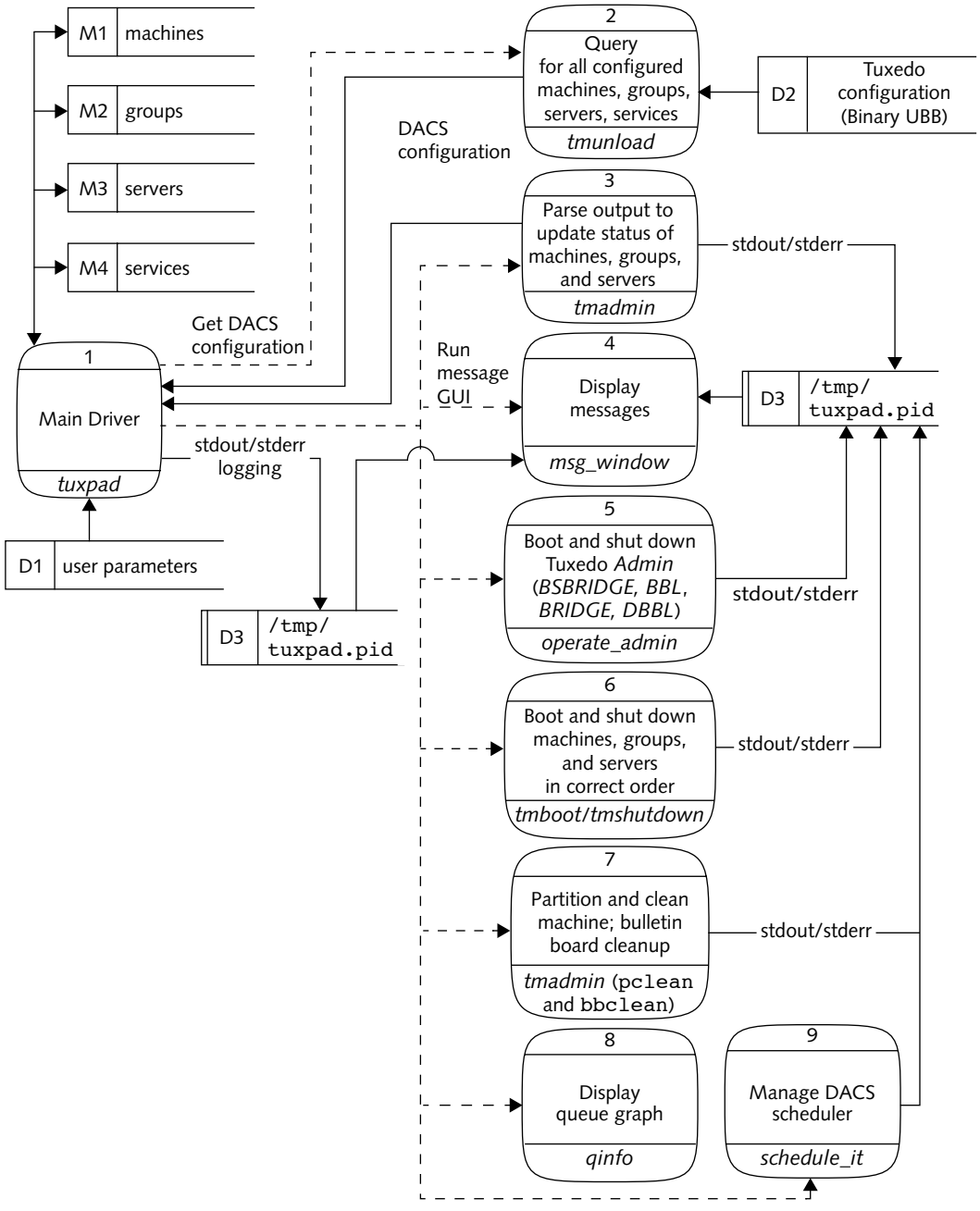


FIGURE 29. TUXPAD DESIGN

operate_admin is a separate or compartmentalized *tuxpad* function, that performs the shut down (`tmshutdown`) and boot (`tmboot`) of the Tuxedo administrative servers (*BSBRIDGE, BRIDGE, BBL, DBBL*) for all Tuxedo DACS machines (process 5 in Figure 29). *operate_admin* boots the machines in the order they appear in the UBB configuration and shuts them down in the reverse order.

msg_window provides a GUI for the display of messages, warnings and errors that are produced by *tuxpad*, *schedule_it*, and *qinfo*. The GUI presents the messages in a scrolling window that can be cleared via a button press. The total number of buffered messages is also displayed. *msg_window* is designed around a UNIX `tail` command that is issued on the *tuxpad* temporary logging file created by *tuxpad* (process 4 in Figure 29). *tuxpad* redirects standard output and standard error to the temporary file so that all output by *tuxpad* and any other program or script that is started by *tuxpad* (for example, *schedule_it*) is captured and displayed. *msg_window* is started by a *tuxpad* button and is intended to run via *tuxpad*.[14]

qinfo provides a GUI to display the state of a Tuxedo qspace. The script is a convenient front end to the Tuxedo *qmadmin* queue administration utility (Figure 30). *qinfo* runs *qmadmin* on the specified QHOST. The QHOST can be reset within *tuxpad* so that the backup qspace can also be monitored via a separate *qinfo* instance. *qinfo* dynamically updates the display at a user-defined interval by presenting the colored bars to show the number of messages in each queue. *qinfo* issues the *qmadmin* commands and parses command output to open the qspace (command *qopen*) and obtains the name and number of messages queued in every queue in the qspace (command *qinfo*). The qspace and queues that are monitored by *qinfo* are defined by user parameters where each queue name to be monitored is specified along with the color to use for the message queue length graph.

14. *msg_window* could be run stand-alone, however the *tuxpad* temporary file name would have to be known, which is possible but not convenient to determine.

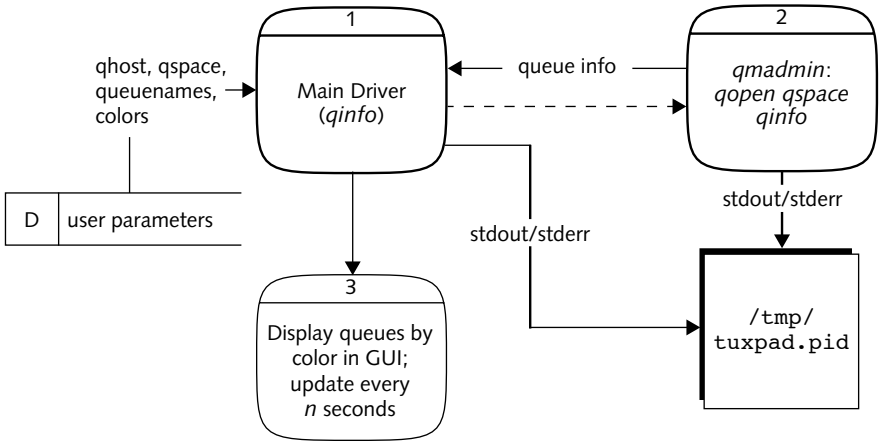


FIGURE 30. QINFO DESIGN

schedule_it provides a GUI to display and manipulate the scheduling system's schedule service table. The script is a convenient front end to the *schedclient* command (Figure 31). *schedule_it* issues *schedclient* commands and parses results from *schedclient*. The *schedclient* commands supported by *schedule_it* are as follows:

- *show* – for on-demand querying and displaying of the service list
- *stall* and *unstall* – for stalling or unstalling user-selected service(s)
- *init* – for re-initializing the scheduling system
- *kick* – for resetting the scheduling system

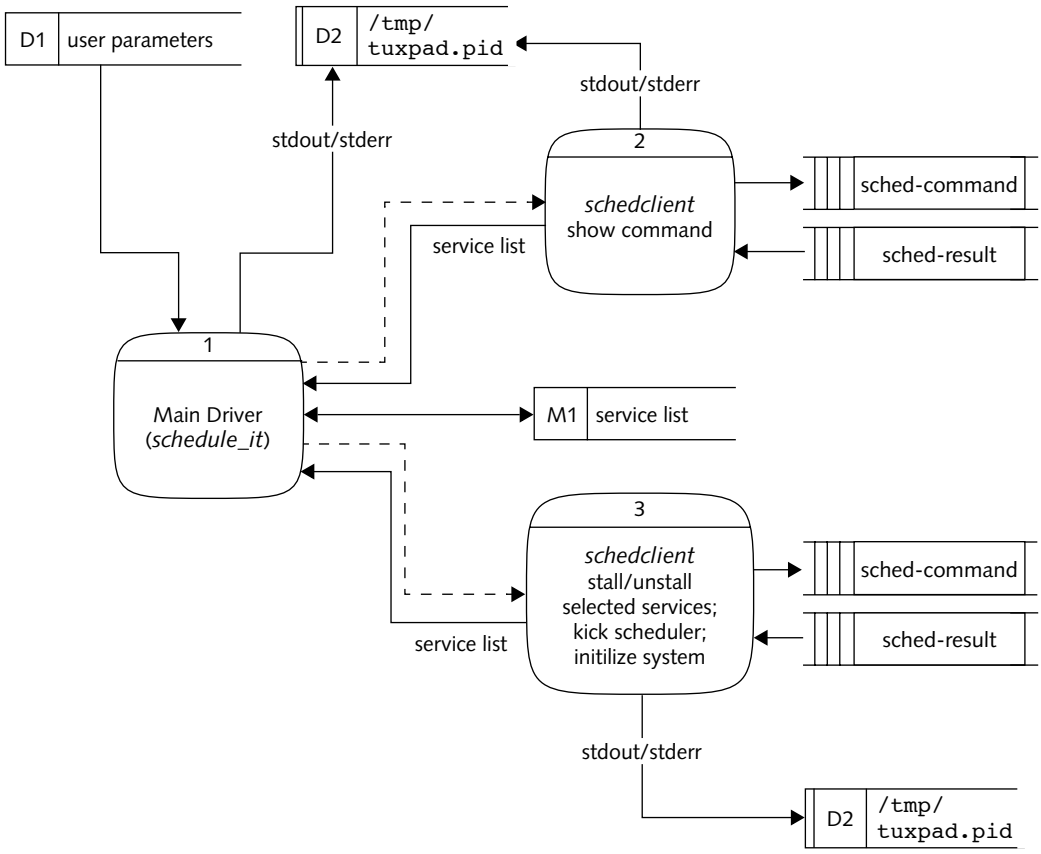


FIGURE 31. SCHEDULE_IT DESIGN

Input/Processing/Output

tuxpad receives input from user parameters, the Tuxedo and DACS administrative commands and clients that it executes, and from the user via manipulations and selections on the GUI. User parameters define various optional user preferences, the Tuxedo master host (THOST), and the primary Tuxedo queuing server (QHOST). The user parameters also include pointers to all standard system variables (for example, IMSPAR), which are required for successful execution of the

Tuxedo and DACS commands. Machine, group, and server booting and shut down, carried out via the *tmadmin* command, must be executed on the THOST, and as such *tuxpad* must also be run on the THOST.

During *tuxpad* initialization, internal arrays of configured machines, groups, servers, and services are created by parsing the output of the *tmunloadcf* command. This command produces/returns an ASCII text version of the entire distributed application configuration. These arrays are central to all supported *tuxpad* operations (M1–M4 in Figure 29 on page 112). The arrays are updated automatically on a user-specified interval or more typically on demand following operator selection of the refresh (R) GUI button. The arrays are updated through a parsing of the *tmadmin* command, which outputs the current state of the distributed application. The current state of the DACS is returned/displayed on the *tuxpad* main display with the presentation organized by user selection of the GUI-provided scrolled vertical lists of machine, group, or server. A color-coded number is displayed adjacent to each listed machine, group, or server. The number represents the number of elements running (number of machines, groups, and servers) where a value other than `0` or `1` is most relevant for servers, which can be configured to run many replicated copies. The color red denotes shut down, green denotes running, and yellow represents running where the number running is not the configured maximum.

tuxpad is designed to drive all operator tasks for system start up and system maintenance. Initial system booting, system shut down, and all intermediate machine, group, or server booting and shut down are handled via *tuxpad*-driven *tmboot* and *tmshutdown* commands. The commands are built on the fly and target specific machines, group, or servers selected by the user through the *tuxpad* GUI. Administrative recovery or cleanup from machine failures is accomplished through *tmadmin* executions using the *tmadmin pclean* and *bbclean* sub-commands. The *tuxpad* output outside the main GUI window consists of output messages and errors generated by the executed commands (for example., *tmboot, qinfo, schedule_it*). The output from the commands is captured by *tuxpad* and redirected to the *tuxpad* temporary output file that is written to `/tmp/tuxpad.`<*tuxpad_pid*> on the local machine. The output is visible to the operator, provided the *msg_window* script is running so that the message window GUI is displayed.

qinfo receives input from user parameters, the *qmadmin* utility following execution of the command, and the user via GUI selections (see Figure 30 on page 114). The user parameters define the QHOST, and qspace name that is to be opened and queried. Parameters also define the complete list of queues that are to be queried and parsed to determine the current number of messages stored in each queue. This list includes specification of the color graph that is output by *qinfo* in the GUI. User input is limited to control of the vertical scroll bar, which enables output of any queue plots that are not presently visible on screen. *qinfo* errors are directed to the *tuxpad* message GUI window as described above.

schedule_it receives input from user parameters, *schedclient* (following execution of the *schedclient show* command), and the user via GUI selections (see Figure 31 on page 115). The user parameters are limited to the file path name of the *schedclient* user parameter file, which is used for every *schedclient* command generated and run by *schedule_it*. *schedule_it* is built around the scheduling system's service list, which is stored in an internal array. This array is central to all supported *tuxpad* operations (M1 in Figure 31 on page 115). The array is initialized and updated by parsing the output of the *schedclient show* command. The parsed input consists of a list of service names including the scheduled time for the next service call and the configured delay time. *schedule_it* displays this service list in the GUI. Selections of one or more services can be checked by the operator to define specific services to *stall* or *unstall* using the *schedclient stall* or *unstall* commands.

schedule_it is primarily designed to provide a simple and direct front end to *schedclient*. However, like *tuxpad*, *schedule_it* is also designed to support some more sophisticated compound command sequences. An operator selection of the Kick Sched button results in the *kick* command sent to *schedclient*, but only after stalling all services in the service list via the *stall schedclient* command for each service. *schedule_it* errors are directed to the *tuxpad* message GUI window as described above.

The GUIs for *tuxpad*, *schedule_it*, *qinfo*, and *msg_window* are implemented using the Tk windowing toolkit, which is accessible via the interpreted Perl/Tk scripting language. The GUI design and layout relies upon widgets for a main form, upon which buttons, scroll bars, text lists and text input boxes are constructed in GUI widget hierarchy specific to each script/GUI.

Control

tuxpad is typically started by the operator and usually through a system-wide start script such as *start_Tuxpad*. *tuxpad* should be run on the THOST for complete access to all features and must be run as the Tuxedo DACS user (UID) that has permission to run the commands *tmadmin, qmadmin*, and so on. *qinfo* can be run stand-alone or, more typically, is started by *tuxpad* following operator selection of the Q info button. *tuxpad* takes care to remote execute *qinfo* on the QHOST machine, which is essential because the *qmadmin* command must be run on the QHOST. *schedule_it* can also be run stand-alone but is also usually run following operator selection of the Scheduler button. The same holds true for *msg_window*, which is displayed following operator selection of the Msg Window button. All *tuxpad* scripts are terminated following operator selection of the Exit buttons on each respective GUI.

Interfaces

Data exchange among *tuxpad*, *operate_admin, schedule_it*, *qinfo*, and *msg_window* is primarily file based. *tuxpad* updates machine, group, and server status by parsing the standard file output returned from a run of *tmadmin*. *schedule_it* and *qinfo* work along the same lines by parsing standard file output from *schedclient* and *qmadmin* respectively. *msg_window* updates the GUI message window with any new output written to the `/tmp/tuxpad.pid` file by *tuxpad*, *schedule_it*, or *qinfo*.

Data exchange within *tuxpad*, *schedule_it,* and *qinfo* is based upon memory stores. These memory stores maintain dynamic lists of machines, groups, and servers in the case of *tuxpad*, queues in the case of *qinfo*, and scheduled services in the case of *schedule_it*.

Error States

tuxpad, *operate_admin*, *schedule_it*, *qinfo*, and *msg_window* are for the most part front ends to the Tuxedo administrative commands and the *schedclient* DACS command. These commands are generated in well-known constructions, and as such not many error states are directly associated with the scripts. Error states resulting from exercising the scripts and options selectable within the GUI can and do result in errors. The breadth of the error states is substantial because *tuxpad* controls and administers a distributed application. The discussion of general system errors is beyond the scope of this document. However, the *tuxpad msg_window* GUI provides a convenient capture of error messages that can be used by the operator to initiate system remedies. "Operator Interventions" on page 65, "Maintenance" on page 89, and "Troubleshooting" on page 101 of [IDC6.5.2Rev0.1], can be used as a source for debugging the DACS error states.

DATABASE DESCRIPTION

This section describes the database design, database entity relationships, and database schema required by the DACS. The DACS relies on the database for all aspects of **interval** creation, updating, and monitoring. Management of the **interval** table involves access to several other database tables. The DACS also reads and updates the **request** table. Access to the database is made through *libgdi*.

Database Design

The entity-relationship model of the schema is shown in Figure 32. The database design for the DACS is based upon the **interval** table, where one **interval** record is created by the DACS for each Automatic Processing pipeline and for each defined interval of time. The **interval**.*state* column is updated by the DACS to reflect the processing state or pipeline sequence as each interval is processed. Station-based pipeline processing is driven by **wfdisc** records, which are read to determine any newly acquired station channel waveforms that have not yet been processed. Static **affiliation** records are read to map a network (*net*) name to a list of stations, to map a station to a list of station sites, to map a station site to a list of station

channels. The **request** table is read and updated in a manner similar to the **interval** table, except that **request** records are only read and updated and are not created by the DACS. The **interval** records are indexed by a unique identifier, stored in the *intvlid* column, and the **lastid** table is read and updated to retrieve and assign unique identifiers for each new **interval** record. The **timestamp** table is used to store the progress of **interval** creation by time. The **timestamp** records are managed for most of the processing pipelines, where the last successful **interval** creation for the pipeline is recorded. The **timestamp** records are also used to store the current **wfdisc**.*endtime* on a station-by-station basis. Updates to these **timestamp** records are handled by the database triggers *wfdisc_endtime* and *wfdisc_NVIAR_endtime*. Application of the triggers allows substantial performance gains when trying to query **wfdisc**.*endtime* by station, because there are very few records in the **timestamp** table compared to the **wfdisc** table.

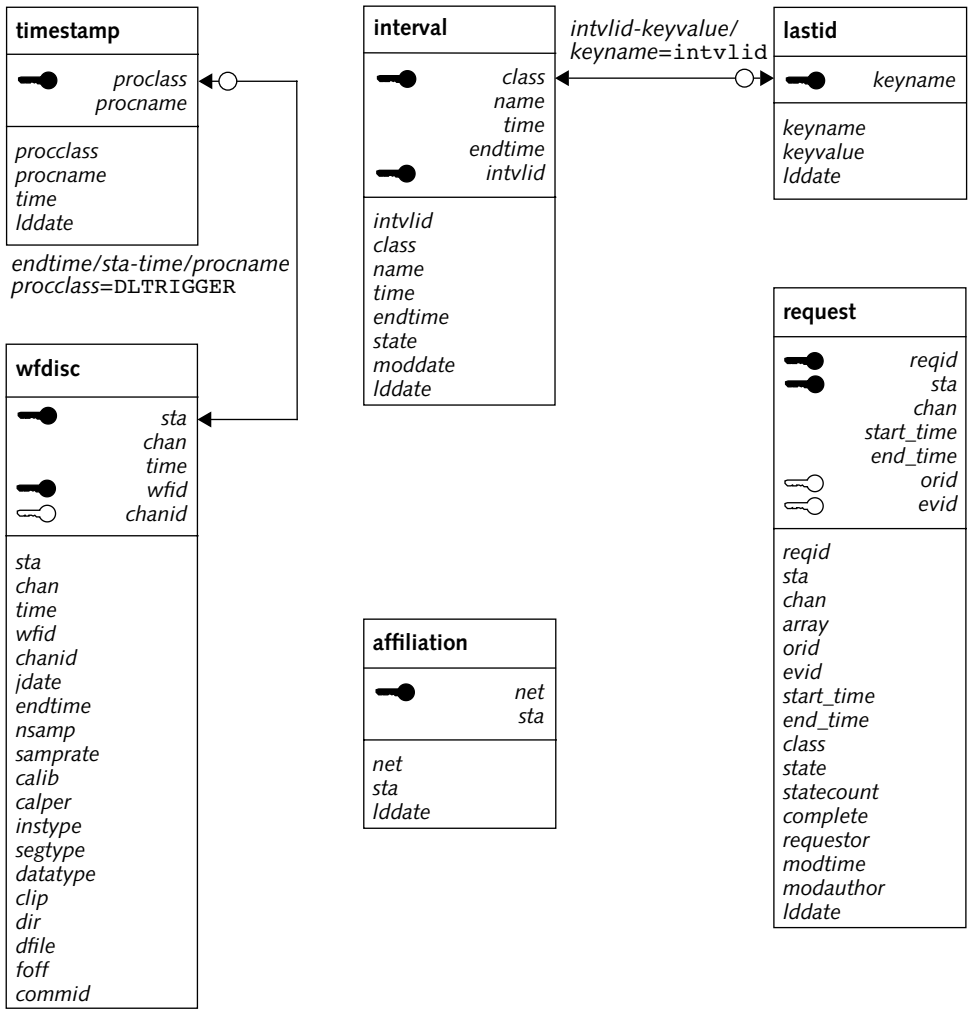


FIGURE 32. ENTITY RELATIONSHIP OF SAIC DACS CSCS

Database Schema

Table 5 shows the database tables used by the DACS. For each table used, the third column shows the purpose for reading or writing each attribute for each relevant CSC.

TABLE 5: DATABASE USAGE BY DACS

Table	Action	Usage by CSC
affiliation	read	*tis_server*, *tiseg_server*: *net* and *sta* are read to map a network name to station names and station name to station sites.
interval	read, write	Data Monitors:[1] *time, endtime, class, state, and intvlid* are read, created, and updated by the interval creation algorithms. *dbserver*: *state* is updated via *tuxshell* service calls to *dbserver*. *WorkFlow*: **interval** records are read and displayed graphically, and *state* is updated as part of interval reprocessing.
lastid	read, write	Data Monitors:[1] key value and *keyname* are used to ensure unique *intvlid's* for each interval creation via a *lock-for-update* database operation.
request	read, write	*WaveGet_server: array, chan, start_time, end_time, state, statecount,* and *requestor* are used to prioritize and request auxiliary waveform acquisition or terminate repeated and unsatisfied requests. *tiseg_server*: *array, start_time, end_time,* and *state* are used to initiate auxiliary station processing for requests that are complete as defined by *state*.[2] *dbserver*: *state* is updated via *tuxshell* service calls to *dbserver*. *WorkFlow*: **request** records are read and displayed graphically.

TABLE 5: DATABASE USAGE BY DACS (CONTINUED)

Table	Action	Usage by CSC
timestamp	read, write	Data Monitors:[1] *procclass, procname,* and *time* are used to track interval creation progress and retrieve current **wfdisc** station *endtime.* *wfdisc_endtime*, *wfdisc_NVIAR_endtime*: These databases trigger update *time* upon any **wfdisc**.*endtime* update.
wfdisc	read	*tis_server*, *tiseg_server*: *time*, *endtime*, *sta*, *chan* are used to determine data availability for continuous and auxiliary data stations.

1. Data Monitors include *tis_server, tiseg_server, ticron_server, tin_server, and WaveGet_server.*
2. The IDC does not use the **request**-based interval creation feature of *tiseg_server.*

Chapter 5: Requirements

This chapter describes the requirements of the DACS and includes the following topics:

- Introduction
- General Requirements
- Functional Requirements
- CSCI External Interface Requirements
- CSCI Internal Data Requirements
- System Requirements
- Requirements Traceability

Chapter 5: Requirements

INTRODUCTION

The requirements of the DACS can be categorized as general, functional, or system requirements. General requirements are nonfunctional aspects of the DACS. These requirements express goals, design objectives, and similar constraints that are qualitative properties of the system. The degree to which these requirements are actually met can only be judged qualitatively. Functional requirements describe what the DACS is to do and how it is to do it. System requirements pertain to general constraints, such as compatibility with other IDC subsystems, use of recognized standards for formats and protocols, and incorporation of standard subprogram libraries.

GENERAL REQUIREMENTS

The DACS capabilities derive from the twin needs to manage the processes in the system and to add an additional layer of fault tolerance. The process management includes starting, stopping, monitoring, communicating and tasking (assigning work). The fault tolerance includes reconfiguring Automatic Processing in the event of a computer failure.

The DACS shall provide the following modes in support of Automatic Processing and Interactive Processing: shutdown, stop, fast-forward, play, slow-motion, rewind, and pause. Table 6 describes the modes. Fast-forward mode (catch-up mode) is configured to add more front-end automatic processing when recovering from a significant time period of complete data outage or system down-time. Rewind mode allows for reprocessing of the most recent data by resetting the database to an earlier time.[1] Pause mode allows current processing tasks to finish prior to a shutdown of the system.

TABLE 6: DACS OPERATIONAL MODES

Requirement Number	Mode	Automatic Processing	Interactive Processing
1.	shutdown	no automatic processing, DACS not running	no interactive processing, DACS not running
2.	stop	no automatic processing, all automatic processing system status saved in stable storage, all automatic processing programs terminated, all DACS processes idle	full interactive processing
3.	fast-forward	full automatic processing, automatic processing configured for burst data (for example, GA replaced by additional instances of *DFX*)	full interactive processing
4.	play	full automatic processing, automatic processing configured for normal operation	full interactive processing
5.	slow-motion	partial automatic processing, automatic processing configured to run only a core subset of automatic processing tasks	full interactive processing
6.	rewind	full automatic processing after resetting the database to an earlier time	full interactive processing
7.	pause	completion of active automatic processing	full interactive processing

1. Slow-motion is used to maintain time-critical automatic processing when the full processing load exceeds the processing capacity.

An additional general requirement is:

8. The DACS shall be started at boot time by a computer on the IDC local area network. The boot shall leave the DACS in the stop state. After it is in this state, the DACS shall be operational and unaffected by the halt or crash of any single computer on the network.

FUNCTIONAL REQUIREMENTS

This section provides specific requirements for the DACS. Each subparagraph describes a group of related requirements. The requirements are grouped into the functional categories of availability management, message passing, workflow management, system monitoring, and reliability.

Availability Management

Availability management refers to the availability of UNIX processes. An availability manager is a service that starts and stops processes according to predefined rules and on-the-fly operator decisions. The rules usually specify a certain number of processes to keep active; if one should terminate then a replacement is to be started.

9. The DACS shall be capable of starting and stopping any configured user-level process on any computer in the IDC LAN. The DACS shall provide an interface to an operator that accepts process control commands. A single operator interface shall allow process control across the network.

10. The DACS shall maintain (start and restart) a population of automated and interactive processes equal to the number supplied in the DACS configuration file. The DACS shall also monitor its internal components and maintain them as necessary.

11. The DACS shall start and manage processes upon messages being sent to a named service. If too few automated processes are active with the name of the requested service, the DACS shall start additional processes (up to a limit)

that have been configured to provide that service. If an interactive process is not active, the DACS shall start a single instance of the application when a message is sent to that application.

12. The DACS shall be fully operational in stop mode within 10 minutes of network boot.

13. The DACS shall detect process failures within 30 seconds of the failure and server hardware failures within 60 seconds.

14. The DACS shall start new processes and replace failed processes within five seconds. This time shall apply to both explicit user requests and the automatic detection of a failure.

15. The DACS shall be capable of managing (starting, monitoring, terminating) 50 automated and interactive processing programs on each of up to 50 computers.

16. The DACS shall continue to function as an availability manager in the event of defined hardware and software failures. "Reliability" on page 134 specifies the DACS reliability and continuous operations requirements.

Message Passing

Message passing in the context of the DACS refers to the transmission of messages between cooperating interactive applications. Message passing is a service provided by the DACS to processes that operate outside the scope of the DACS. The DACS does not interpret or otherwise operate on the message.

17. The DACS shall provide a message passing service for the interactive processing system. The message passing service shall have the attributes of being reliable, asynchronous, ordered, scoped, point-to-point, and location transparent. The message passing service shall provide an API to the interactive processing programs. Each attribute is specified in the following subparagraph.

17.1 Reliable: messages are not lost and no spurious messages are created. A consequence of reliable messages is that the same message may be delivered more than once if a process reads a message, crashes, restarts, then reads a message again.

17.2 Asynchronous: sending and receiving processes need not be running or communicating concurrently.

17.3 Ordered: messages are delivered in the order they were sent (FIFO).

17.4 Scoped: messages sent and received by one interactive user are not crossed with messages sent and received by another user.

17.5 Point-to-point: There is a single sender and a single receiver for each message. The DACS need not support broadcast or multicast, although sending processes may simulate either by iteratively sending the same message to many receivers (one-to-many). Similarly, many-to-one messaging is supported by multiple point-to-point messaging, that is, receiving processes may receive separate messages from many senders.

17.6 Location transparency: sending and receiving processes do not need to know the physical location of the other. All addressing of messages is accomplished through logical names.

17.7 Application programming interface: the message service will be available to the Interactive Processing programs via a software library linked at compile time.

18. The message passing service shall provide an administrative control process to support administrative actions. The administrative actions shall allow a user to add or delete messages from any message queue and to obtain a list of all processes registered to receive messages.

19. The DACS shall deliver messages within one second of posting given that network utilization is below 10 percent of capacity.

20. If the receiving process is not active or is not accepting messages, the DACS shall hold the message indefinitely until delivery is requested by the receiving process (or deleted by an administrative control process).

21. Interactive processing programs may request the send or receive of messages at any time. Multiple processes may simultaneously request any of the message services.
22. The DACS shall be capable of queuing (holding) 10,000 messages for each process that is capable of receiving messages.
23. The size limit of each message is 4,096 (4K) bytes in length.
24. The DACS shall continue to function as a message passing service in the event of defined hardware and software failures. The DACS reliability and continuous operations requirements are specified in "Reliability" on page 134.

Workflow Management

Workflow management in the context of the DACS refers to the marshalling of data through data processing sequences. The steps (tasks) in a data processing sequence are independent of each other with the exception of order. That is, if step B follows step A, then step B may be initiated any time after the successful termination of step A. The independence of the processing tasks allows task B to run on a different computer than task A.

Workflow management allows for different types of ordering. Sequential ordering requires that one task run before another task. Parallel ordering allows two tasks to execute simultaneously, yet both must finish before the next task in the sequence may begin. Conditional ordering allows one of two tasks to be selected as the next task in the sequence based on the results of the current processing task. Finally, a compound ordering allows for a sub-sequence of tasks within a task sequence. A compound statement requires all internal processing steps to finish before the next interval is submitted to the compound statement.

25. The DACS shall provide workflow management for the Automatic Processing. Workflow management ensures that data elements get processed by a sequence of Automatic Processing programs. A data element is a collection of data, typically a discrete time interval of time-series data, that is maintained by processes external to the DACS. The DACS workflow management shall cre-

ate, manage, and destroy internal references to data elements. The DACS references to data elements are known as intervals. The capabilities of the workflow management are enumerated in the following subparagraphs.

25.1 The DACS shall provide a configurable method of defining data elements. The parametric definition of data elements shall include at least a minimum and maximum time range, a percentage of data required, a list of channels/stations, and a percentage of channels and/or stations required. If the data in an interval are insufficient to meet the requirements for an interval, then the data element shall remain unprocessed. In this case, the DACS shall identify the interval as insufficient and provide a means for the operator to manually initiate a processing sequence.

25.2 The DACS shall provide a configurable method of initiating a workflow sequence. The DACS workflow management shall be initiated upon either data availability, completion of other data element sequences, or the passage of time.

25.3 Workflow management shall allow sequential processing, parallel processing, conditional branching, and compound statements.

25.4 Workflow management shall support priority levels for data elements. Late arriving or otherwise important data elements may be given a higher priority so that they receive priority ordering for the next available Automatic Processing program. Within a single priority group, the DACS shall manage the order among data elements by attributes of the data, including time and source, and by attributes of the interval, including elapsed time in the queue. The ordering algorithm shall be an option to the operator.

25.5 Workflow management shall provide error recovery per data element for failures of the Automatic Processing programs. Error recovery shall consist of a limited number of time-delayed retries of the failed Automatic Processing program. If the retry limit is reached, the DACS shall hold the failed intervals in a failed queue for manual intervention.

25.6 The DACS shall initiate workflow management of each data element within 5 seconds of data availability.

25.7 Workflow management shall deliver intervals from one Automatic Processing program to the next program in the sequence within five seconds of completion of the first program. If the second program is busy with another interval, the workflow management shall queue the interval and deliver the interval with the highest priority in the queue within 5 seconds of when the second program becomes available.

26. The DACS shall be capable of queuing (holding) 10,000 intervals for each active Automatic Processing program (there can be up to fifty processes per computer). The size and composition of an interval is left as a detail internal to the DACS.

27. The DACS shall continue to function as a workflow manager in the event of defined hardware and software failures. The DACS reliability and continuous operations requirements are specified in "Reliability" on page 134.

System Monitoring

System monitoring in the context of the DACS refers to monitoring of DACS-related computing resources. System monitoring does not include monitoring of operating systems, networks, or hardware except for the detection and workaround of computer crashes.

28. The DACS shall provide system monitoring for computer status, process status, workflow status, and the message passing service.

29. The DACS shall monitor the status of each computer on the network, and the status of all computers shall be visible on the operator's console, current to within 30 seconds.

30. The DACS shall provide an interface to indicate the run-time status of all processes relevant to Automatic Processing and Interactive Processing. This set of processes includes database servers and DACS components.

30.1 The DACS shall provide a display indicating the last completed automatic processing step for each interval within the workflow management.

30.2 The same display shall provide a summary that indicates the processing sequence completion times for all intervals available to Interactive Processing (that is, more recent than the last data migration).

31. The DACS shall provide a graphical display of the status of message passing with each Interactive Processing program. The status shall indicate the interactive processes capable of receiving messages and whether there are any messages in the input queue for each receiving process.

32. The DACS displays shall remain current within 60 seconds of actual time. The system monitoring displays shall provide a user-interface command that requests an update of the display with the most recent status.

33. The DACS run-time status display shall be capable of displaying all processes managed by the availability manager. The DACS message passing display shall be capable of displaying the empty/non-empty message queue status of all processes that can receive messages. The DACS workflow management display shall be capable of displaying all intervals currently managed by the workflow management.

34. The DACS shall provide these displays simultaneously to 1 user, although efforts should be made to accommodate 10 additional users.

35. The DACS shall continue to function as a system monitor in the event of defined hardware and software failures. The DACS reliability and continuous operations requirements are described in "Reliability" on page 134.

Reliability

Reliability in the context of the DACS refers primarily to the integrity of the workflow management and message passing, and secondarily to the continued (but perhaps limited) operation of the DACS during system failures. The DACS is one of the primary providers of computing reliability in the IDC System.

The integrity of the DACS guarantees that messages are delivered exactly once, and Automatic Processing is invoked exactly once for each data element. Messages and data sequences are preserved across system failures. When forced to choose, the DACS takes the conservative approach of preserving data at the expense of timely responses.

The DACS provides continued operation in the event of defined system failures. The DACS operation may be interrupted briefly as replacement components are restarted, possibly on other computers. The DACS monitors and restarts both internal components and Automatic Processing programs. Interactive programs are not restarted because it is not known whether the user intentionally terminated a program.

36. The DACS shall deliver each message exactly once, after the successful posting of the message by the sending process.

37. The DACS shall execute Automatic Processing programs exactly once for each data element. A program execution is a transaction consisting of start, run, and exit. If the transaction aborts before completion of the exit, the DACS shall retry the transaction a limited (configurable) number of times.

38. The DACS shall function as a system in the event of defined hardware and software failures. The failure model used by the DACS is given in Table 7. For failures within the model, the DACS shall mask and attempt to repair the failures. Failure masking means that any process depending upon the services of the DACS (primarily the Automatic and Interactive Processing software) remains unaffected by failures other than to notice a time delay for responses from the failed process. Failures outside the failure model may lead to undefined behavior (for example, a faulty ethernet card is undetectable and unrepairable by software).

39. The DACS shall detect failures and respond to failures within specified time limits. The time limits are given in Table 7.

40. The DACS shall detect and respond to failures up to a limited number of failures. The failure limits are given in Table 7. For failures over the limit, the DACS shall attempt the same detection and response, but success is not guaranteed.

41. Reliability of a system or component is relative to a specified set of failures listed in Table 7. The first column indicates the types of failures that the DACS shall detect and recover from. The second column lists the maximum rate of failures guaranteed to be handled properly by the DACS; however, the DACS shall strive to recover from all errors of these types regardless of frequency. The third column lists the upper time bounds on detecting and recovering from the indicated failures. Again, the DACS shall strive to attain the best possible detection and recovery times.

TABLE 7: FAILURE MODEL

No.	Failure Type	Maximum Failure Rate	Maximum Time to Recover
41.1	workstation crash failure	one per hour, non-overlapping	60 seconds for detection and 5 seconds to initiate recovery
41.2	process crash failure	five per hour, onset at least 5 minutes apart	5 seconds for detection and 5 seconds to initiate recovery
41.3	process timing failure–all but interactive applications	five per hour, onset at least 5 minutes apart	5 seconds for detection and 5 seconds to initiate recovery
41.4	process timing failure–interactive applications	not detectable	user detection and recovery
41.5	all others	undefined	undefined

CSCI EXTERNAL INTERFACE REQUIREMENTS

The DACS shall have four direct external interfaces and shall operate on an assumed model of data availability. The interfaces are specified in the following paragraphs.

The DACS interfaces with the Database Management System through the GDI, with the operator through an operator interface, with the Interactive Processing through a messaging interface, and with the host operating system through system utilities. The exact data model exported by the Database Management System is critical to the DACS.

42. The DACS shall interface with the ORACLE database through the GDI.

43. The DACS shall read from the **wfdisc** table. The DACS shall assume **wfdisc** table entries will follow the data model described in [IDC5.1.1Rev2].

44. The DACS shall insert and update entries in the **interval** table, which is used as a monitoring point for the Automatic Processing system. As part of reset mode, the DACS may delete or alter entries in the **interval** table to force reprocessing of recent data elements. Purging of the **interval** table is left to processes outside the DACS.

45. The DACS shall interface with the **wfdisc** table of the ORACLE database. The software systems of the Data Services SCSI shall acquire the time-series data and populate the **wfdisc** table. The DACS shall assume a particular model for **wfdisc** record insertion and updates. The DACS shall be capable of accepting data in the model described by the following subparagraphs.

    45.1 The IDC Continuous Data system acquires seismic, hydroacoustic, and infrasonic waveforms from multiple sources. The data quantity is 5–10 gigabytes of data per day arriving in a near-continuous fashion. The DACS nominally forms intervals of segments of 10 minutes in length. However, during recovery of a data acquisition system failure, the DACS forms intervals of up to one hour in length. The DACS can be configured to form intervals of practically any size.

45.2 The data from each source nominally arrive in piecewise increasing time order. Data delivery from an individual station may be interrupted and then resumed. Upon resumption of data delivery, the data acquisition system may provide current data, late data, or both. Current data resumes with increasing time, and late data may fill in a data gap in either increasing FIFO or decreasing LIFO time order from the end points of the time gap.

Figure 33 shows an example where current (continuous) data are interrupted and then resumed, which is then followed by examples of both FIFO and LIFO late data arrival. In (A) continuous data arrive with advancing time. (B) Data are interrupted; no data arrive. (C) Data begin to arrive again starting with the current time. (D) Both late data and continuous data arrive in tandem; the late data fills in the data gap in FIFO order. (E) Both late data and continuous data arrive in tandem; the late data fill in the data gap in LIFO order.

The data acquisition system defines each channel of a seismic station, array, hydroacoustic sensor, or infrasonic sensor as a separate data source. The result is that some channels may be delivered later than other channels from the same station or the channels might not be delivered at all.

45.3 Data quality is a prime concern of the IDC mission; however, the DACS makes no determination of data quality. Any data that are available shall be processed.

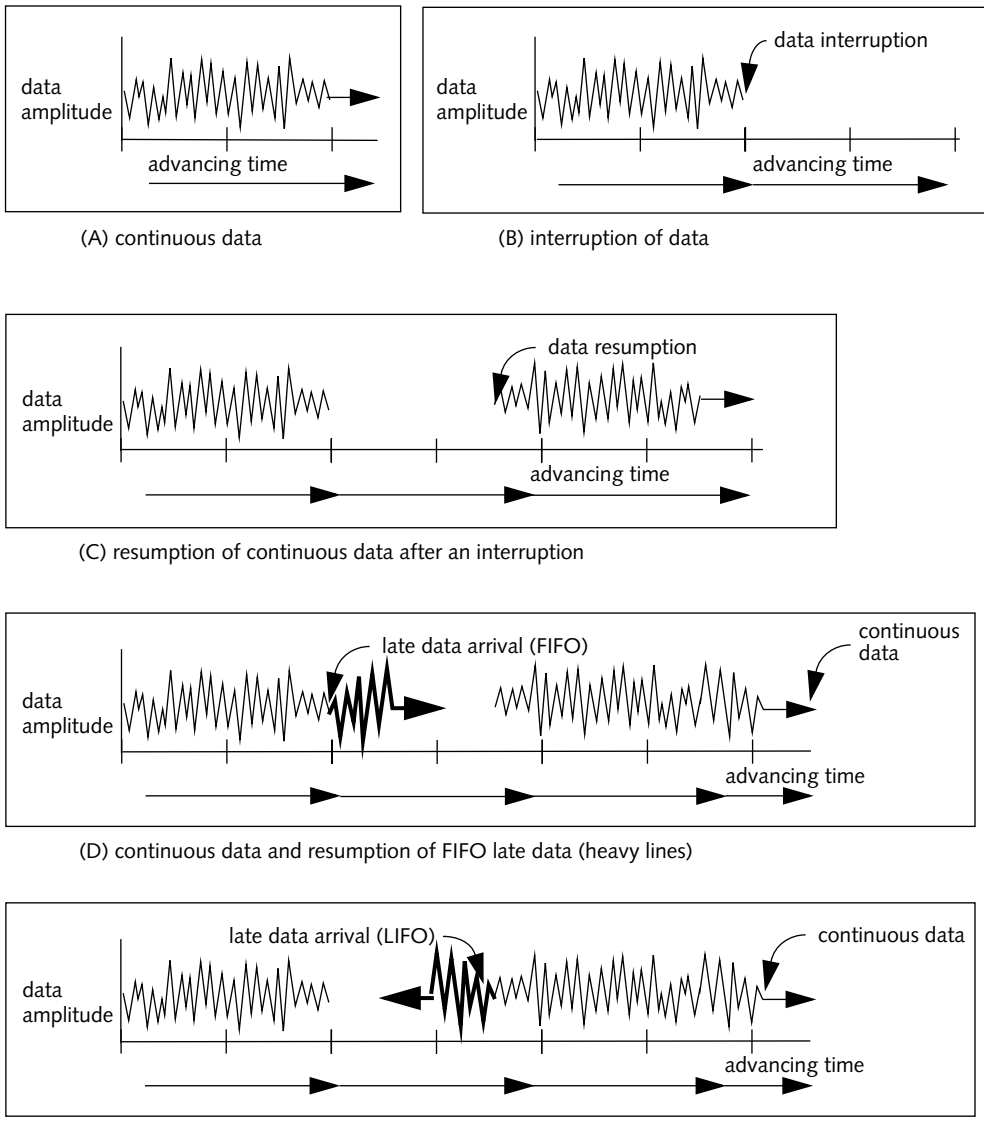


FIGURE 33. DATA ARRIVAL EXAMPLE

46. The DACS shall interface with the Interactive Processing programs through a message passing API. The DACS shall provide this interface as a library for use by the developers of the Interactive Processing programs. The library shall contain entry points to allow processes to register, subscribe, unregister, send, poll, receive, replay and delete messages. The DACS shall offer several types of notification when new messages are sent to a process. The API is specified in more detail in the following list.
    - 46.1 register–connect to messaging system; arguments specify logical name and physical location of process; method of notification for waiting messages
    - 46.2 subscribe–specify types of messages to read; argument lists message types to read
    - 46.3 unregister–disconnect from messaging system; argument indicates whether to keep or discard unread messages
    - 46.4 send–send a message to another process by logical name; arguments specify message type, message data, and return address of sender
    - 46.5 poll–request empty/non-empty status of incoming message queue
    - 46.6 receive–receive a message; argument specifies message types to read
    - 46.7 delete–delete messages from queue; argument specifies most recent or all messages
47. The DACS shall offer three types of notification of new messages: none, callback invocation, or an interrupt. The type shall be chosen by a process when it registers. With none, the process shall call the poll function to check on message availability. With callback invocation, the process shall register a callback procedure to be executed when a message arrives. With an interrupt, the process shall rely on the interrupt (such as activity on a UNIX file descriptor) to indicate when a message is waiting.
48. The DACS shall interface with the UNIX operating system to start Automatic Processing programs and wait on the termination of these programs. Processes started by the DACS shall inherit the system privileges of the DACS, including the process group, environment, and file system permissions.

49. The DACS shall collect the exit or abnormal termination status of processes it starts. The exit status shall be used to determine success or failure of the Automatic Processing program. Processes shall use a defined set of exit codes to indicate various levels of success and another set of codes to indicate different types of failure.

50. The DACS shall interface with an operator or operators. The DACS shall provide monitoring displays and control interfaces. The monitoring displays shall provide system monitoring for computer status, process status, workflow status, and the message passing service. (The information presented with each monitoring display is specified in "System Monitoring" on page 133.) The control interface shall enable the operator to take actions on the DACS. The control interface supports the functions listed in the following subparagraphs.

    50.1 The DACS control interface shall allow selection from among the automatic processing modes listed in Table 6 on page 127.

    50.2 The DACS control interface shall allow run-time reconfiguration of the host computer network. Reconfiguration may take the form of added, deleted, or upgraded workstations. The DACS shall allow an operator to dynamically identify the available workstations. When a workstation is removed from service, the DACS shall migrate all processes on that workstation to other workstations. The time allowed for migration shall be the upper run-time limit for the Automatic Processing programs. In other words, running programs shall be allowed to complete before the migration occurs.

    50.3 The DACS control interface shall allow run-time reconfiguration of the DACS programs. Reconfiguration shall allow an increase, decrease, or migration of Automatic Processing programs.

    50.4 The DACS control interface shall allow access to the availability manager for starting or stopping individual DACS and Automatic Processing programs.

    50.5 The DACS control interface shall allow manual processing and reprocessing of data elements through their respective sequences.

51. The DACS shall acquire time from a global time service.

CSCI INTERNAL DATA REQUIREMENTS

52. The DACS shall maintain a collection of intervals (data element references) and shall update the status of intervals in the **interval** database table.

SYSTEM REQUIREMENTS

The DACS shall be configurable.

53. The implementation of the DACS shall allow for configuration data of the number and type of computers on the network, and the number of automated processes of each type allowed to execute on each computer type.

    The implementation of the DACS also requires the execution parameters for each process in the Automated and Interactive Processing.

54. Only authorized users shall be allowed to initiate processing. Unauthorized requests shall be rejected and logged. The DACS shall require passwords from authorized users at login.

55. The DACS shall operate in the IDC environment.

56. The DACS shall operate in the same hardware environment as the IDC.

57. The DACS requires extensive database queries to detect new wfdisc records. These queries will impact the database server. Otherwise, the DACS shall consume negligible hardware resources.

58. Similarly, the DACS must share the same software environment as the rest of the IDC. While this environment is not exactly defined at this time, it is likely to include:

    - Solaris 7 or 8
    - ORACLE 8.x
    - X Window System X11R5 or later
    - TCP/IP network utilities

59. The DACS shall adhere to ANSI C, POSIX, and SQL standards.

60. The DACS shall use common UNIX utilities (for example, cron, sendmail) and system calls (for example, sockets, exec) whenever possible, to take advantage of widespread features that shall aid portability. Vendor-specific UNIX utilities shall be isolated into separate modules for identification and easy replacement should the need arise.
61. The DACS shall implement middleware layers to isolate third-party software products and protocol standards.
62. The DACS shall implement the functions of workflow management, availability management, inter-process communications, and system monitoring as separate stand-alone programs.
63. The DACS shall use COTS for internal components where practical. Practical in this situation means where there is a strong functional overlap between the DACS requirements and COTS capabilities.
64. The DACS shall be designed to scale to a system twice as large as the initial IDC requirements without a noticeable degradation in time to perform the DACS functions.
65. The DACS requires a capable UNIX system administrator for installation of the DACS components and system-level debugging of problems such as file system full, insufficient UNIX privileges, and network connectivity problems.
66. The DACS shall be delivered with a System Users Manual that explains the operations and run-time options of the DACS. The manual shall also specify all configuration parameters of the DACS. The DACS shall only require a user-level prior understanding of UNIX and Motif.
67. The DACS shall be delivered electronically.
68. The DACS capabilities of workflow management and message passing are ranked equally high in terms of criticality. These capabilities shall function in the event of system failures. The functions of availability management and system monitoring rank next in order of importance. The DACS shall continue to perform the first set of functions even if the second set of functions are unavailable for any reason.

REQUIREMENTS TRACEABILITY

Tables 8 through 16 trace the requirements of the DACS to components and describe how the requirements are fulfilled.

TABLE 8: TRACEABILITY OF GENERAL REQUIREMENTS

	Requirement	How Fulfilled
1	Operational Mode: shutdown Automatic Processing: no automatic processing, DACS not running Interactive Processing: no interactive processing, DACS not running	For Automatic Processing the DACS can be shutdown under operator control using *tuxpad* (scripts: *tuxpad* and *schedule_it*) or a Tuxedo administration utility and *schedclient*. For Interactive Processing this requirement is fulfilled the same as for Automatic Processing although in practice the operators tend not to have to administer the DACS because it automatically starts on machine boot and normally requires zero administration. The *crInteractive* script is also used by the operator to administer Interactive Processing instance(s).
2	Operational Mode: stop Automatic Processing: no automatic processing, all automatic processing system status saved in stable storage, all automatic processing programs terminated, all DACS processes idle Interactive Processing: full interactive processing	For Automatic Processing the DACS can be stopped under operator control using *tuxpad* (scripts *tuxpad* and *schedule_it*) or a Tuxedo administration utility and *schedclient*. In the stop mode, all of the DACS is terminated except for the Tuxedo administration servers (for example, BBL) on each DACS machine. For Interactive Processing this requirement is fulfilled the same as above and also normally is never required.

TABLE 8: TRACEABILITY OF GENERAL REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
3	Operational Mode: fast-forward Automatic Processing: full automatic processing, automatic processing configured for burst data (for example, GA replaced by additional instances of *DFX*) Interactive Processing: full interactive processing	For Automatic Processing the DACS provides extensive support for scaling the number of machines, servers, and services as well as such resources that are active at any given time. Fast-forward can be displayed (via *tuxpad*) by deactivating or shutting down one type of server and activating or booting another type of server(s) (for example, GA replaced by additional instances of *DFX*). For Interactive Processing this requirement is fulfilled the same as above, although this processing mode is not generally applicable to interactive processing.
4	Operational Mode: play Automatic Processing: full automatic processing, automatic processing configured for normal operation Interactive Processing: full interactive processing	For Automatic Processing the play processing mode is usually initiated by starting the scheduling of the data monitor servers. This is accomplished via the `kick` command to the scheduling system typically using the *tuxpad schedule_it* script. For Interactive Processing the play processing mode is the default and automatic processing mode following the DACS startup (following analyst workstation boot).

TABLE 8: TRACEABILITY OF GENERAL REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
5	Operational Mode: slow-motion Automatic Processing: partial automatic processing, automatic processing configured to run only a core subset of automatic processing tasks Interactive Processing: full interactive processing	For Automatic Processing the DACS provides extensive support for scaling the number of machines, servers, and services as well as which of these resources are active at any given time. Slow-motion can be displayed (via *tuxpad*) by deactivating or shutting down a class or servers (for example, network processing) or reducing the number of a particular type of server (for example, reduce the number of *DFX* instances). In addition, the *tuxpad schedule_it* script can be used to stall data monitor instances to eliminate or reduce the creation of new pipeline processing sequences. For Interactive Processing this requirement is fulfilled the same as above although this processing mode is not generally applicable to Interactive Processing.
6	Operational Mode: rewind Automatic Processing: full automatic processing after resetting the database to an earlier time Interactive Processing: full interactive processing	For Automatic Processing the rewind processing mode requires an operator to delete intervals in the **interval** table or set them to *state* `skipped` where applicable so that data monitor servers will completely reprocess a time period of data.[1] For Interactive Processing this mode is not applicable as far as the DACS is concerned. Repeated Event Review is controlled by the analyst.

TABLE 8: TRACEABILITY OF GENERAL REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
7	Operational Mode: pause Automatic Processing: completion of active automatic processing Interactive Processing: full interactive processing	For Automatic Processing the pause mode is displayed by stalling scheduling of the data monitor servers using the *tuxpad schedule_it* script and possibly the shutdown of the DACS *TMQFORWARD* servers to stop processing of queued intervals. For Interactive Processing this requirement is fulfilled the same as above although this processing mode is not generally applicable to interactive processing.
8	The DACS shall be started at boot time by a computer on the IDC local area network. The boot shall leave the DACS in the stop state. After it is in this state, the DACS shall be operational and unaffected by the halt or crash of any single computer on the network.	The DACS is booted by the operator usually via *tuxpad*, and the DACS is effectively in the stop or pause mode awaiting operator action to initiate the play mode. The DACS can survive the crash of a single computer in most cases. Single points of failure include the database server and the file logging server, which are accepted single points of failure. The scheduling system queue server is a single point of failure. This single point of failure can be masked by migrating the scheduling queue server to an existing machine that is a single point of failure such as the database server or file logging server.

1. The rewind mode is also partially addressed by operator-assisted interval reprocessing by *WorkFlow*. Full automatic reprocessing could be provided by the *WorkFlow* reprocessing model by augmenting the existing scheme to support reprocessing of all intervals or all intervals of a particular class for a specified range of time. However, this feature would have to be consistent with the fact that application software must be able to repeat the processing steps. Furthermore, reprocessing is also subject to IDC policy decisions, particularly where intermediate or final processing results have been published or made available as IDC products.

TABLE 9: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: AVAILABILITY MANAGEMENT

	Requirement	How Fulfilled
9	The DACS shall be capable of starting and stopping any configured user-level process on any computer in the IDC LAN. The DACS shall provide an interface to an operator that accepts process control commands. A single operator interface shall allow process control across the network.	Any DACS process can be started or stopped by the operator using *tuxpad* or a Tuxedo administration utility.
10	The DACS shall maintain (start and restart) a population of automated and interactive processes equal to the number supplied in the DACS configuration file. The DACS shall also monitor its internal components and maintain them as necessary.	Complete process monitoring including boot and shutdown of all configured processes as well as monitoring and restart of all configured processes is provided by the DACS via Tuxedo.
11	The DACS shall start and manage processes upon messages being sent to a named service. If too few automated processes are active with the name of the requested service, the DACS shall start additional processes (up to a limit) that have been configured to provide that service. If an interactive process is not active, the DACS shall start a single instance of the application when a message is sent to that application.	For Automatic Processing the Tuxedo DACS generally starts servers and keeps them running, so server startup upon message send is not typically required. However, server scaling is supported wherein the number of active servers advertising a given service name can increase as the number of queued messages increases. For Interactive Processing the *dman* client supports demand execution, which starts a single application instance upon a message send if the application is not already running.
12	The DACS shall be fully operational in stop mode within 10 minutes of network boot.	For Automatic Processing the DACS can take several minutes to completely boot across the LAN but the time does not exceed 10 minutes. For Interactive Processing the DACS boots in approximately 30 seconds.

TABLE 9: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: AVAILABILITY MANAGEMENT (CONTINUED)

	Requirement	How Fulfilled
13	The DACS shall detect process failures within 30 seconds of the failure and server hardware failures within 60 seconds.	The DACS can be configured to detect server and machine failures well within the required specification. The configuration is via the Tuxedo `ubbconfig` file.
14	The DACS shall start new processes and replace failed processes within five seconds. This time shall apply to both explicit user requests and the automatic detection of a failure.	Same as above.
15	The DACS shall be capable of managing (starting, monitoring, terminating) 50 automated and interactive processing programs on each of up to 50 computers.	The DACS can scale to the required specification and beyond.
16	The DACS shall continue to function as an availability manager in the event of defined hardware and software failures. "Reliability" on page 134 specifies the DACS reliability and continuous operations requirements.	The DACS continues to function or can be configured to function in the face of most process and system failures. Exceptions include failure of the database server and file logging server machines, which are accepted single points of failure.

TABLE 10: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: MESSAGE PASSING

	Requirement	How Fulfilled
17	The DACS shall provide a message passing service for the interactive processing system. The message passing service shall have the attributes of being reliable, asynchronous, ordered, scoped, point-to-point, and location transparent. The message passing service shall provide an API to the interactive processing programs. Each attribute is specified in the following subparagraph.	The message passing requirements are fulfilled by the DACS *libipc* API. Location transparency (messaging across machine or via the LAN) is fully supported but not generally used at the IDC.[1]
17.1	Reliable: messages are not lost and no spurious messages are created. A consequence of reliable messages is that the same message may be delivered more than once if a process reads a message, crashes, restarts, then reads a message again.	This requirement is fulfilled via *libipc* messaging, which is based on the Tuxedo reliable queuing service.
17.2	Asynchronous: sending and receiving processes need not be running or communicating concurrently.	This requirement is fulfilled via *libipc* messaging, which is based on the Tuxedo reliable queuing service.
17.3	Ordered: messages are delivered in the order they were sent (FIFO).	This requirement is fulfilled via *libipc* messaging which is based on the Tuxedo reliable queuing service.
17.4	Scoped: messages sent and received by one interactive user are not crossed with messages sent and received by another user.	This requirement is fulfilled via *libipc* messaging, which is based on the Tuxedo reliable queuing service. Message scoping is supported via queue names that are scoped to application name and session number. Multiple analysts running a single machine would have to run in their own sessions. In general the operational model is one analyst per machine and it is up to analysts to manage their own sessions within a single machine.

TABLE 10: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: MESSAGE PASSING (CONTINUED)

	Requirement	How Fulfilled
17.5	Point-to-point: There is a single sender and a single receiver for each message. The DACS need not support broadcast or multicast, although sending processes may simulate either by iteratively sending the same message to many receivers (one-to-many). Similarly, many-to-one messaging is supported by multiple point-to-point messaging, that is, receiving processes may receive separate messages from many senders.	All messaging is point-to-point but with the required asychronous delivery wherein the Tuxedo queuing system is the reliable message broker. There is limited and specific support for event/message broadcasting, where *libipc* sends an event broadcast to the DACS *dman* client for each message send and receive within the interactive session. The *dman* client also subscribes to Tuxedo event broadcasts, which announce the joining and departing of a client of the interactive session.
17.6	Location transparency: sending and receiving processes do not need to know the physical location of the other. All addressing of messages is accomplished through logical names.	This requirement is fulfilled via *libipc* messaging, which is based on the Tuxedo reliable queuing service.
17.7	Application programming interface: the message service will be available to the Interactive Processing programs via a software library linked at compile time.	This requirement is fulfilled via *libipc* messaging, which is based on the Tuxedo reliable queuing service.
18	The message passing service shall provide an administrative control process to support administrative actions. The administrative actions shall allow a user to add or delete messages from any message queue and to obtain a list of all processes registered to receive messages.	This requirement is fulfilled by the *birdie* client, which is a driver to test *libipc*. Most of these requirements, among others, are also fulfilled by the *dman* client. With *dman*, the analyst can delete all messages but not individual messages. Message addition is supported through message sends from specific Interactive Tools within the interactive session.

TABLE 10: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: MESSAGE PASSING (CONTINUED)

	Requirement	How Fulfilled
19	The DACS shall deliver messages within one second of posting given that network utilization is below 10 percent of capacity.	Reliable queue messaging (disk and transaction based messaging) within the DACS can occur at least 10 times per second.
20	If the receiving process is not active or is not accepting messages, the DACS shall hold the message indefinitely until delivery is requested by the receiving process (or deleted by an administrative control process).	This requirement is fulfilled via *libipc* messaging, which is based on the Tuxedo reliable queuing service.
21	Interactive processing programs may request the send or receive of messages at any time. Multiple processes may simultaneously request any of the message services.	This requirement is fulfilled via *libipc* messaging, which is based on the Tuxedo reliable queuing service.
22	The DACS shall be capable of queuing (holding) 10,000 messages for each process that is capable of receiving messages.	This requirement is fulfilled via the Tuxedo reliable queuing service which can be scaled well beyond the specification.
23	The size limit of each message is 4,096 (4K) bytes in length.	This requirement is fulfilled via the Tuxedo reliable queuing service, which can be scaled beyond the specification.[2]
24	The DACS shall continue to function as a message passing service in the event of defined hardware and software failures. The DACS reliability and continuous operations requirements are specified in "Reliability" on page 134.	This requirement is fulfilled via the DACS' ability to survive most failure conditions as discussed previously.

1. Interactive Processing is configured to run on a stand-alone analyst machine; all Interactive Tools and messages reside on a single machine.
2. The maximum message size was increased to 65,536 bytes for the Interactive Auxiliary Data Request System. This increase deviates from the model of passing small referential data between processes for both Interactive and Automatic Processing. The change was made specifically for Interactive Processing. This change encourages a re-examination of the messaging requirements: message size, message reliability, and so on.

TABLE 11: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: WORKFLOW MANAGEMENT

	Requirement	How Fulfilled
25	The DACS shall provide workflow management for the Automatic Processing. Workflow management ensures that data elements get processed by a sequence of Automatic Processing programs. A data element is a collection of data, typically a discrete time interval of time-series data, that is maintained by processes external to the DACS. The DACS workflow management shall create, manage, and destroy internal references to data elements. The DACS references to data elements are known as intervals. The capabilities of the workflow management are enumerated in the following subparagraphs.	This requirement is fulfilled in the DACS by a number of components and features including: reliable queuing, transactions, process monitoring, data monitor servers, *tuxshell*, and so on.
25.1	The DACS shall provide a configurable method of defining data elements. The parametric definition of data elements shall include at least a minimum and maximum time range, a percentage of data required, a list of channels/stations, and a percentage of channels and/or stations required. If the data in an interval are insufficient to meet the requirements for an interval, then the data element shall remain unprocessed. In this case, the DACS shall identify the interval as insufficient and provide a means for the operator to manually initiate a processing sequence.	This requirement is fulfilled by the DACS data monitor servers, specifically *tis_server* and *tiseg_server,* and the ability to specify the required parameters related to interval creation.
25.2	The DACS shall provide a configurable method of initiating a workflow sequence. The DACS workflow management shall be initiated upon either data availability, completion of other data element sequences, or the passage of time.	This requirement is fulfilled by the DACS data monitor servers and the ability to specify the required parameters related to interval creation.

TABLE 11: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: WORKFLOW MANAGEMENT (CONTINUED)

	Requirement	How Fulfilled
25.3	Workflow management shall allow sequential processing, parallel processing, conditional branching, and compound statements.	This requirement for sequential processing and compound processing is fulfilled by the DACS process sequencing function (*TMQFORWARD* and *tuxshell[s]*). Distributed parallel processing is achieved in part by configuring or replicating like servers across machines and/or across processors within a machine. Parallel processing pipelines or sequences, and conditional branching, are fulfilled through the use of data monitor servers. Data monitor server instances create new pipeline sequences as a function of specified availability criteria. As such, parallel pipelines are broken or decomposed into multiple sub-pipelines where each sub-pipeline is created by a specific data monitor server instance. There is no supported mechanism within Tuxedo DACS to specify and process a complex pipeline processing sequence as one parameter or one process sequence expression or function.
25.4	Workflow management shall support priority levels for data elements. Late arriving or otherwise important data elements may be given a higher priority so that they receive priority ordering for the next available Automatic Processing program. Within a single priority group, the DACS shall manage the order among data elements by attributes of the data, including time and source, and by attributes of the interval, including elapsed time in the queue. The ordering algorithm shall be an option to the operator.	This requirement is fulfilled via the DACS data monitor support for priority-based queuing and related support for interval creation that gives preference to late arriving or otherwise important data.[1] Operator access to this support is through data monitor parameter files.

TABLE 11: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: WORKFLOW MANAGEMENT (CONTINUED)

	Requirement	How Fulfilled
25.5	Workflow management shall provide error recovery per data element for failures of the Automatic Processing programs. Error recovery shall consist of a limited number of time-delayed retries of the failed Automatic Processing program. If the retry limit is reached, the DACS shall hold the failed intervals in a failed queue for manual intervention.	This requirement is fulfilled by the DACS *tuxshell* server.
25.6	The DACS shall initiate workflow management of each data element within 5 seconds of data availability.	Reliable queue messaging (disk- and transaction-based messaging) within the DACS can occur at least 10 times per second, and workflow management of each data element can be initiated with the same frequency. However, *tis_server* database queries currently take about 20 seconds at the IDC, and *tis_server* is currently configured to run every 90 seconds. Therefore, the worst case is in excess of 100 seconds after data are available. The 5 second requirement is not possible given the current database-server dependence.
25.7	Workflow management shall deliver intervals from one Automatic Processing program to the next program in the sequence within five seconds of completion of the first program. If the second program is busy with another interval, the workflow management shall queue the interval and deliver the interval with the highest priority in the queue within 5 seconds of when the second program becomes available.	Same as above.

TABLE 11: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: WORKFLOW MANAGEMENT (CONTINUED)

	Requirement	How Fulfilled
26	The DACS shall be capable of queuing (holding) 10,000 intervals for each active Automatic Processing program (there can be up to fifty processes per computer). The size and composition of an interval is left as a detail internal to the DACS.	This requirement is fulfilled via the Tuxedo reliable queuing service, which can be scaled well beyond the specification.
27	The DACS shall continue to function as a workflow manager in the event of defined hardware and software failures. The DACS reliability and continuous operations requirements are specified in "Reliability" on page 134.	This requirement is fulfilled via the DACS' ability to survive most failure conditions, as discussed previously.

1. This feature has been at least partially implemented but has not been sufficiently tested to date.

TABLE 12: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: SYSTEM MONITORING

	Requirement	How Fulfilled
28	The DACS shall provide system monitoring for computer status, process status, workflow status, and the message passing service.	This requirement is fulfilled in the DACS through Tuxedo, *WorkFlow*, *tuxpad*, and *dman* for the DACS clients and servers.
29	The DACS shall monitor the status of each computer on the network, and the status of all computers shall be visible on the operator's console, current to within 30 seconds.	This requirement is fulfilled in the DACS through Tuxedo, *WorkFlow*, *tuxpad*, and *dman* for DACS clients and servers.

TABLE 12: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: SYSTEM MONITORING (CONTINUED)

	Requirement	How Fulfilled
30	The DACS shall provide an interface to indicate the run-time status of all processes relevant to Automatic Processing and Interactive Processing. This set of processes includes database servers and DACS components.	This requirement is fulfilled in the DACS through *tuxpad* and *dman*, but the database server is not monitored because this is not a DACS process.
30.1	The DACS shall provide a display indicating the last completed automatic processing step for each interval within the workflow management.	This requirement is fulfilled by the *WorkFlow* application.
30.2	The same display shall provide a summary that indicates the processing sequence completion times for all intervals available to Interactive Processing (that is, more recent than the last data migration).	Same as above.
31	The DACS shall provide a graphical display of the status of message passing with each Interactive Processing program. The status shall indicate the interactive processes capable of receiving messages and whether there are any messages in the input queue for each receiving process.	This requirement is fulfilled by the *dman* client.
32	The DACS displays shall remain current within 60 seconds of actual time. The system monitoring displays shall provide a user-interface command that requests an update of the display with the most recent status.	This requirement is fulfilled in general because the DACS is always processing in real time or near real time. Specifically, the DACS status at the machine or server level is available in real time via the *tuxpad* refresh button. *WorkFlow* updates on an operator-specified update interval or on demand via a GUI selection.

TABLE 12: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: SYSTEM MONITORING (CONTINUED)

	Requirement	How Fulfilled
33	The DACS run-time status display shall be capable of displaying all processes managed by the availability manager. The DACS message passing display shall be capable of displaying the empty/non-empty message queue status of all processes that can receive messages. The DACS workflow management display shall be capable of displaying all intervals currently managed by the workflow management.	This requirement is fulfilled by *tuxpad*, *dman*, *qinfo*, and *WorkFlow*.
34	The DACS shall provide these displays simultaneously to 1 user, although efforts should be made to accommodate 10 additional users.	Any number of users logged in as the "Tuxedo" user can access *tuxpad*. Typically, *dman* would only be accessed by the analyst that is using the interactive session that *dman* is managing. *WorkFlow* can be viewed by any number of users.
35	The DACS shall continue to function as a system monitor in the event of defined hardware and software failures. The DACS reliability and continuous operations requirements are described in "Reliability" on page 134.	This requirement is fulfilled via the DACS' ability to survive most failure conditions, as discussed previously.

TABLE 13: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: RELIABILITY

	Requirement	How Fulfilled
36	The DACS shall deliver each message exactly once, after the successful posting of the message by the sending process.	This requirement is fulfilled via the Tuxedo reliable queuing service, which uses transactions to ensure that each message is delivered only once.

TABLE 13: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: RELIABILITY (CONTINUED)

	Requirement	How Fulfilled
37	The DACS shall execute Automatic Processing programs exactly once for each data element. A program execution is a transaction consisting of start, run, and exit. If the transaction aborts before completion of the exit, the DACS shall retry the transaction a limited (configurable) number of times.	This requirement is fulfilled by the DACS' *TMQFORWARD* server and transaction.
38	The DACS shall function as a system in the event of defined hardware and software failures. The failure model used by the DACS is given in Table 7. For failures within the model, the DACS shall mask and attempt to repair the failures. Failure masking means that any process depending upon the services of the DACS (primarily the Automatic and Interactive Processing software) remains unaffected by failures other than to notice a time delay for responses from the failed process. Failures outside the failure model may lead to undefined behavior (for example, a faulty ethernet card is undetectable and unrepairable by software).	This requirement is fulfilled via the DACS ability to survive most failure conditions as discussed previously.
39	The DACS shall detect failures and respond to failures within specified time limits. The time limits are given in Table 7.	This requirement is fulfilled via the DACS' ability to survive most failure conditions, as discussed previously. No time limits have been specified, but the DACS can be configured to service most failure conditions and recover from them in less than 10 seconds.

TABLE 13: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: RELIABILITY (CONTINUED)

	Requirement	How Fulfilled
40	The DACS shall detect and respond to failures up to a limited number of failures. The failure limits are given in Table 7. For failures over the limit, the DACS shall attempt the same detection and response, but success is not guaranteed.	This requirement is fulfilled via the DACS' ability to survive most failure conditions, as discussed previously. Or, if this requirement refers to application failures, these failures are handled as described by *tuxshell*.
41	Reliability of a system or component is relative to a specified set of failures listed in Table 7. The first column indicates the types of failures that the DACS shall detect and recover from. The second column lists the maximum rate of failures guaranteed to be handled properly by the DACS; however, the DACS shall strive to recover from all errors of these types regardless of frequency. The third column lists the upper time bounds on detecting and recovering from the indicated failures. Again, the DACS shall strive to attain the best possible detection and recovery times.	This requirement is fulfilled via the DACS' ability to survive most failure conditions as discussed previously.
41.1	workstation crash failure Maximum Failure Rate: one per hour, non-overlapping Maximum Time to Recover: 60 seconds for detection and 5 seconds to initiate recovery	This requirement is fulfilled via the DACS' ability to survive a workstation failure subject to the DACS being configured with sufficient backup servers to survive a single machine failure. The specified detection and recovery times can be met through configuration of the `ubbconfig` file.
41.2	process crash failure Maximum Failure Rate: five per hour, onset at least 5 minutes apart Maximum Time to Recover: 5 seconds for detection and 5 seconds to initiate recovery	Same as above but at the process level.

TABLE 13: TRACEABILITY OF FUNCTIONAL REQUIREMENTS: RELIABILITY (CONTINUED)

	Requirement	How Fulfilled
41.3	process timing failure–all but interactive applications Maximum Failure Rate: five per hour, onset at least 5 minutes apart Maximum Time to Recover: 5 seconds for detection and 5 seconds to initiate recovery	To the extent that this requirement refers to process time-outs, it is fulfilled through *tuxshell's* support for child-process time-out management. Otherwise, all process failures are detected and automatically recovered by the DACS as discussed previously.
41.4	process timing failure–interactive applications Maximum Failure Rate: not detectable Maximum Time to Recover: user detection and recovery	In general the analyst detects and recovers from these failures. The DACS for Interactive Processing does include process monitoring and time-out monitoring for *tuxshell* child processes.
41.5	all others Maximum Failure Rate: undefined Maximum Time to Recover: undefined	N/A

TABLE 14: TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS

	Requirement	How Fulfilled
42	The DACS shall interface with the ORACLE database through the GDI.	All DACS access to the database server is through the *GDI*.
43	The DACS shall read from the wfdisc table. The DACS shall assume wfdisc table entries will follow the data model described in [IDC5.1.1Rev2].	The DACS data monitor applications *tis_server* and *tiseg_server* read the **wfdisc** table. Access to the table is fully compatible with the published database schema.

TABLE 14: TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
44	The DACS shall insert and update entries in the interval table, which is used as a monitoring point for the Automatic Processing system. As part of reset mode, the DACS may delete or alter entries in the interval table to force reprocessing of recent data elements. Purging of the interval table is left to processes outside the DACS.	The DACS manages the **interval** table to reflect the state of all automatic processing. Interval deletion is not generally supported, which is apparently not a problem. Intervals are changed as a part of interval reprocessing accessible through *WorkFlow*.
45	The DACS shall interface with the wfdisc table of the ORACLE database. The software systems of the Data Services SCSI shall acquire the time-series data and populate the wfdisc table. The DACS shall assume a particular model for wfdisc record insertion and updates. The DACS shall be capable of accepting data in the model described by the following subparagraphs.	The DACS reads the **wfdisc** table. Access to the table is fully compatible with the published database schema.
45.1	The IDC Continuous Data system acquires seismic, hydroacoustic, and infrasonic waveforms from multiple sources. The data quantity is 5–10 gigabytes of data per day arriving in a near-continuous fashion. The DACS nominally forms intervals of segments of 10 minutes in length. However, during recovery of a data acquisition system failure, the DACS forms intervals of up to one hour in length. The DACS can be configured to form intervals of practically any size.	This requirement is fulfilled through the DACS' ability to specify parameters for variable interval sizes under varying conditions to *tis_server*.

TABLE 14: TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
45.2	The data from each source nominally arrive in piecewise increasing time order. Data delivery from an individual station may be interrupted and then resumed. Upon resumption of data delivery, the data acquisition system may provide current data, late data, or both. Current data resumes with increasing time, and late data may fill in a data gap in either increasing FIFO or decreasing LIFO time order from the end points of the time gap.	*tis_server* can handle all described types of data delivery and can create intervals in the order of current data first.
45.3	Data quality is a prime concern of the IDC mission; however, the DACS makes no determination of data quality. Any data that are available shall be processed.	DACS does not consider data quality as a criteria for interval creation.

TABLE 14: TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
46	The DACS shall interface with the Interactive Processing programs through a message passing API. The DACS shall provide this interface as a library for use by the developers of the Interactive Processing programs. The library shall contain entry points to allow processes to register, subscribe, unregister, send, poll, receive, replay and delete messages. The DACS shall offer several types of notification when new messages are sent to a process. The API is specified in more detail in the following list.	This requirement is fulfilled by *libipc*, except that the ability to replay messages was not addressed. Message subscription is limited to broadcasts to the *dman* client upon any message send and receive. The message polling implementation was changed due to a problem with Tuxedo "unsolicited message" handling. The problem required heavier weight polling, although the increased polling time was well within the relatively light message timing requirements. The change requires querying the queue to see if a new message has been received. The original implementation relied upon relatively light-weight broadcasts that were sent by *libipc* to the receiving client (the client that was being sent the message). Soliciting broadcast traffic is lighter weight than actually checking the receive queue.
46.1	register–connect to messaging system; arguments specify logical name and physical location of process; method of notification for waiting messages	This requirement is fulfilled via the *ipc_attach() libipc* API call. The physical location of the process is implied or transparent to the messaging system. The method of notification for waiting messages is not addressed by this function.
46.2	subscribe–specify types of messages to read; argument lists message types to read	This requirement is fulfilled specifically for *dman* where *libipc* broadcasts to *dman* upon any message send and receive among clients with the interactive session. A general subscribe mechanism is not provided by *libipc* and is apparently not required. However, Tuxedo supports general publish-subscribe messaging.

TABLE 14: TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
46.3	unregister–disconnect from messaging system; argument indicates whether to keep or discard unread messages	This requirement is fulfilled via the *ipc_detach() libipc* API call, although there is no mechanism to direct discarding of unread messages by this function.
46.4	send–send a message to another process by logical name; arguments specify message type, message data, and return address of sender	This requirement is fulfilled via the *ipc_send() libipc* API call.
46.5	poll–request empty/non-empty status of incoming message queue	This requirement is fulfilled via the *ipc_pending() libipc* API call.
46.6	receive–receive a message; argument specifies message types to read	This requirement is fulfilled via the *ipc_receive() libipc* API call.
46.7	delete–delete messages from queue; argument specifies most recent or all messages	This requirement is fulfilled via the *ipc_purge() libipc* API call.

TABLE 14: TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
47	The DACS shall offer three types of notification of new messages: none, callback invocation, or an interrupt. The type shall be chosen by a process when it registers. With none, the process shall call the poll function to check on message availability. With callback invocation, the process shall register a callback procedure to be executed when a message arrives. With an interrupt, the process shall rely on the interrupt (such as activity on a UNIX file descriptor) to indicate when a message is waiting.	Two of the three types of notification are fulfilled although the second type is fulfilled in a modified form. Message notification type "none" is fulfilled via explicit calls to the *ipc_receive() libipc* API call. Message notification type "callback" is fulfilled via the *ipc_add_xcallback() libipc* API call, except that the registered callback or handler function is called every time. The reason for the change is described in requirement 46. The handler function invokes *ipc_receive()* to check a new message. The handler function is called as part of an X11 timer event callback, which is currently configured to happen every 1/2 second unless the client application cannot be presently interrupted (for example, during a database submit). Message notification type "interrupt" is not supported, and this feature currently is not needed.
48	The DACS shall interface with the UNIX operating system to start Automatic Processing programs and wait on the termination of these programs. Processes started by the DACS shall inherit the system privileges of the DACS, including the process group, environment, and file system permissions.	This requirement is fulfilled by *tuxshell*.

TABLE 14: TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
49	The DACS shall collect the exit or abnormal termination status of processes it starts. The exit status shall be used to determine success or failure of the Automatic Processing program. Processes shall use a defined set of exit codes to indicate various levels of success and another set of codes to indicate different types of failure.	This requirement is fulfilled by *tuxshell*.
50	The DACS shall interface with an operator or operators. The DACS shall provide monitoring displays and control interfaces. The monitoring displays shall provide system monitoring for computer status, process status, workflow status, and the message passing service. (The information presented with each monitoring display is specified in "System Monitoring" on page 133.) The control interface shall enable the operator to take actions on the DACS. The control interface supports the functions listed in the following subparagraphs.	This requirement is fulfilled by the *tuxpad* scripts, *WorkFlow*, and the *dman* client.
50.1	The DACS control interface shall allow selection from among the automatic processing modes listed in Table 6 on page 127.	This requirement is fulfilled by the *tuxpad* scripts *tuxpad* and *schedule_it*. The processing modes are defined in requirements 1–7.

TABLE 14: TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
50.2	The DACS control interface shall allow run-time reconfiguration of the host computer network. Reconfiguration may take the form of added, deleted, or upgraded workstations. The DACS shall allow an operator to dynamically identify the available workstations. When a workstation is removed from service, the DACS shall migrate all processes on that workstation to other workstations. The time allowed for migration shall be the upper run-time limit for the Automatic Processing programs. In other words, running programs shall be allowed to complete before the migration occurs.	Run-time host and server migration is supported by the DACS and is accessible via *tuxpad*. Run-time addition of a workstation is supported if the workstation was defined in the `ubbconfig` file. Presumably the workstation is defined but is "dormant" until an operator decides to migrate or initiate processing on the machine. Unconfigured workstations cannot be added during run-time. (Tuxedo supports this feature, but the DACS does not currently use it).
50.3	The DACS control interface shall allow run-time reconfiguration of the DACS programs. Reconfiguration shall allow an increase, decrease, or migration of Automatic Processing programs.	Run-time server migration is supported by the DACS and is accessible via *tuxpad*.
50.4	The DACS control interface shall allow access to the availability manager for starting or stopping individual DACS and Automatic Processing programs.	This requirement is fulfilled via *tuxpad*.
50.5	The DACS control interface shall allow manual processing and reprocessing of data elements through their respective sequences.	This requirement is fulfilled via the interval reprocessing feature of *WorkFlow,* which is based on the *ProcessInterval* script and *SendMessage* client.

TABLE 14: TRACEABILITY OF CSCI EXTERNAL INTERFACE REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
51	The DACS shall acquire time from a global time service.	This requirement is not met. The DACS relies upon external support for clock synchronization (for example, system cron jobs, which attempt to synchronize all machines clocks on the LAN once per day). Or, the DACS relies on the database server for a single source of time. However, the DACS uses both methods for time synchronization without a consistent criterion.

TABLE 15: TRACEABILITY OF CSCI INTERNAL DATA REQUIREMENTS

	Requirement	How Fulfilled
52	The DACS shall maintain a collection of intervals (data element references) and shall update the status of intervals in the interval database table.	This requirement is fulfilled by various DACS elements including the data monitor servers, *tuxshell*, and *dbserver*.

TABLE 16: TRACEABILITY OF SYSTEM REQUIREMENTS

	Requirement	How Fulfilled
53	The implementation of the DACS shall allow for configuration data of the number and type of computers on the network, and the number of automated processes of each type allowed to execute on each computer type.	This requirement is fulfilled by the `ubbconfig` file and parameter files for each DACS application.

TABLE 16: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
54	Only authorized users shall be allowed to initiate processing. Unauthorized requests shall be rejected and logged. The DACS shall require passwords from authorized users at login.	Administration of the DACS, typically carried out through *tuxpad*, is limited to the "Tuxedo" user or the user that owns the DACS processes defined in the `ubbconfig` file. Password authentication is implicitly handled by the operating system. The DACS has not implemented any authentication specific to the CSCI (Tuxedo offers various options to do so if needed).
55	The DACS shall operate in the IDC environment.	Fulfilled.
56	The DACS shall operate in the same hardware environment as the IDC.	Fulfilled.
57	The DACS requires extensive database queries to detect new wfdisc records. These queries will impact the database server. Otherwise, the DACS shall consume negligible hardware resources.	This requirement has been fulfilled. Even though the Tuxedo-based DACS manifests in a large number of processes spread across the LAN, the processes consume a relatively small amount of computing resources. The expense of the **wfdisc** queries has been partially mitigated through the introduction of database triggers. The database triggers update **wfdisc** end time values in an efficient manner saving similar queries, which would otherwise be submitted against the **wfdisc** table.
58	Similarly, the DACS must share the same software environment as the rest of the IDC. While this environment is not exactly defined at this time, it is likely to include: Solaris 7 or 8 ORACLE 8.x X Window System X11R5 or later TCP/IP network utilities	Fulfilled.

TABLE 16: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
59	The DACS shall adhere to ANSI C, POSIX, and SQL standards.	Fulfilled.
60	The DACS shall use common UNIX utilities (for example, cron, sendmail) and system calls (for example, sockets, exec) whenever possible, to take advantage of widespread features that shall aid portability. Vendor-specific UNIX utilities shall be isolated into separate modules for identification and easy replacement should the need arise.	The DACS limits vendor-specific products to Tuxedo. The DACS makes use of public domain software such as Perl/Tk (Perl with Tk GUI bindings). As such, the requirement is fulfilled.
61	The DACS shall implement middleware layers to isolate third-party software products and protocol standards.	This requirement is fulfilled to a reasonable degree. The interactive messaging library, *libipc*, was implemented with the requirement in mind in that the Tuxedo layer is separated from the general messaging API wherever possible. For Automatic Processing, layering is, in certain cases, challenging because deployment of a Tuxedo application such as the DACS is at the system and user configuration levels.
62	The DACS shall implement the functions of workflow management, availability management, inter-process communications, and system monitoring as separate stand-alone programs.	This requirement is fulfilled to a reasonable degree. *WorkFlow* management is implemented by several cooperating programs. Availability management and system monitoring is handled, in part, by Tuxedo, which relies on a distributed set of servers to carry out this function. Inter-process communications is handled by a variety of programs, libraries, and system resources such as qspace disk files.

TABLE 16: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
63	The DACS shall use COTS for internal components where practical. Practical in this situation means where there is a strong functional overlap between the DACS requirements and COTS capabilities.	This requirement is fulfilled by Tuxedo.
64	The DACS shall be designed to scale to a system twice as large as the initial IDC requirements without a noticeable degradation in time to perform the DACS functions.	This requirement is fulfilled by Tuxedo.
65	The DACS requires a capable UNIX system administrator for installation of the DACS components and system-level debugging of problems such as file system full, insufficient UNIX privileges, and network connectivity problems.	This requirement is fulfilled, although the DACS has matured to the point that a UNIX system administrator is not required for the majority of the DACS installation task.
66	The DACS shall be delivered with a System Users Manual that explains the operations and run-time options of the DACS. The manual shall also specify all configuration parameters of the DACS. The DACS shall only require a user-level prior understanding of UNIX and Motif.	This requirement is fulfilled (see [IDC6.5.2Rev0.1]).
67	The DACS shall be delivered electronically.	This requirement is fulfilled.

TABLE 16: TRACEABILITY OF SYSTEM REQUIREMENTS (CONTINUED)

	Requirement	How Fulfilled
68	The DACS capabilities of workflow management and message passing are ranked equally high in terms of criticality. These capabilities shall function in the event of system failures. The functions of availability management and system monitoring rank next in order of importance. The DACS shall continue to perform the first set of functions even if the second set of functions are unavailable for any reason.	This requirement is fulfilled via the DACS' ability to survive most failure conditions as discussed previously.

References

The following sources supplement or are referenced in this document:

[And96] Andrade, J. M., Carges, M. T., Dwyer, T. J., and Felts, S. D., *The TUXEDO System: Software for Constructing and Managing Distributed Business Applications*, Addison-Wesley Publishing Company, 1996.

[BEA96] BEA Systems, Inc., B*EA TUXEDO Reference Manual*, 1996.

[DOD94a] Department of Defense, "Software Design Description," *Military Standard Software Development and Documentation,* MIL-STD-498, 1994.

[DOD94b] Department of Defense, "Software Requirements Specification," *Military Standard Software Development and Documentation,* MIL-STD-498, 1994.

[Gan79] Gane, C., and Sarson, T., *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

[IDC5.1.1Rev2] Science Applications International Corporation, Veridian Pacific-Sierra Research, *Database Schema, Revision 2*, SAIC-00/3057, PSR-00/TN2830, 2000.

[IDC6.5.1] Science Applications International Corporation, *Interactive Analysis Subsystem Software User Manual,* SAIC-01/3001, 2001.

[IDC6.5.2Rev0.1] Science Applications International Corporation, *Distributed Application Control System (DACS) Software User Manual, Revision 0.1*, SAIC-00/3038, 2000.

Glossary

A

admin server

Tuxedo server that provides interprocess communication and maintains the distributed processing state across all machines in the application. Admin servers are provided as part of the Tuxedo distribution.

AEQ

Anomalous Event Qualifier.

application (DACS, Tuxedo)

System of cooperating processes configured for a specific function to be run in a distributed fashion by Tuxedo. Also used in a more general sense to refer to all objects included in one particular `ubbconfig` file (machines, groups, servers) and associated shared memory resources, qspaces, and clients.

application server

Server that provides functionality to the application.

architecture

Organizational structure of a system or component.

architectural design

Collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.

archive

Single file formed from multiple independent files for storage and backup purposes. Often compressed and encrypted.

ARS

Analyst Review Station. This application provides tools for a human analyst to refine and improve the event bulletin by interactive analysis.

ASCII

American Standard Code for Information Interchange. Standard, unformatted 256-character set of letters and numbers.

B

backup (component)

System component that is provided redundantly. Backups exist on the machine, group, server, and services level. Appropriate backups are configured to seamlessly take over processing as soon as a primary system component fails or becomes unavailable.

beam

(1) Waveform created from array station elements that are sequentially summed after being steered to the direction of a specified azimuth and slowness. (2) Any derived waveform (for example, a filtered waveform).

Beamer

Application that prepares event beams for the notify process and for later analysis.

boot

Action of starting a server process as a memory-resident task. Booting the whole application is equivalent to booting all specified server processes (admin servers first, application servers second).

bulletin

Chronological listing of event origins spanning an interval of time. Often, the specification of each origin or event is accompanied by the event's arrivals and sometimes with the event's waveforms.

C

CCB

Configuration Control Board.

CDE

Common Desktop Environment.

child process

UNIX process created by the *fork* routine. The child process is a snapshot of the parent at the time it called *fork*.

click

Select an element on the screen by positioning the pointer over the element, then pressing and immediately releasing the mouse button.

client

Software module that gathers and presents data to an application; it generates requests for services and receives replies. This term can also be used to indicate the requesting role that a software module assumes by either a client or server process.

command

Expression that can be input to a computer system to initiate an action or affect the execution of a computer program.

Common Desktop Environment

Desktop graphical user interface that comes with SUN Solaris.

component

One of the parts of a system; also referred to as a module or unit.

Computer Software Component

Functionally or logically distinct part of a computer software configuration item, typically an aggregate of two or more software units.

Computer Software Configuration Item

Aggregation of software that is designated for configuration management and treated as a single entity in the configuration management process.

configuration

(1) (hardware) Arrangement of a computer system or components as defined by the number, nature, and interconnection of its parts. (2) (software) Set of adjustable parameters, usually stored in files, which control the behavior of applications at run time.

configuration item

Aggregation of hardware, software, or both treated as a single entity in the configuration management process.

control flow

Sequence in which operations are performed during the execution of a computer program.

COTS

Commercial-Off-the-Shelf; terminology that designates products such as hardware or software that can be acquired from existing inventory and used without modification.

crash

Sudden and complete failure of a computer system or component.

CSC

Computer Software Component.

CSCI

Computer Software Configuration Item.

D

DACS

Distributed Application Control System. This software supports inter-application message passing and process management.

DACS machines

Machines on a Local Area Network (LAN) that are explicitly named in the `*MACHINES` and `*NETWORK` sections of the `ubbconfig` file. Each machine is given a logical reference (see LMID) to associate with its physical name.

daemon

Executable program that runs continuously without operator intervention. Usually, the system starts daemons during initialization. (Example: *cron*).

data flow

Sequence in which data are transferred, used, and transformed during the execution of a computer program.

data monitors

Class of application servers that monitor data streams and data availability, form data intervals, and initiate a sequence of general processing servers when a sufficiently large amount of unprocessed data are found.

dequeue

Remove a message from a Tuxedo queue.

detection

Probable signal that has been automatically detected by the *Detection and Feature Extraction* (*DFX*) software.

DFX

Detection and Feature Extraction. *DFX* is a programming environment that executes applications written in Scheme (known as *DFX* applications).

diagnostic

Pertaining to the detection and isolation of faults or failures.

disk loop

Storage device that continuously stores new waveform data while simultaneously deleting the oldest data on the device.

DM

Data monitor.

dman

Distributed Application Manager. This software element of the DACS manages the availability (execution) of processes.

E

enqueue

Place a message in a Tuxedo queue.

F

failure

Inability of a system or component to perform its required functions within specified performance requirements.

forwarding agent

Application server *TMQFORWARD* that acts as an intermediary between a message queue on disk and a group of processing servers advertising a service. The forwarding agent uses transactions to manage and control its forwarding function.

G

GA

Global Association application. GA associates S/H/I phases to events.

generalized processing server

DACS application server (*tuxshell*) that is the interface between the DACS and the automatic processing system. It executes application programs as child processes.

GUI

Graphical User Interface

H

host

Machine on a network that provides a service or information to other computers. Every networked computer has a hostname by which it is known on the network.

hydroacoustic

Pertaining to sound in the ocean.

I

IDC

International Data Centre.

infrastructure

Foundation and essential elements of a system or plan of operation.

instance

Running computer program. An individual program may have multiple instances on one or more host computers.

IPC

Interprocess communication. The messaging system by which applications communicate with each other through *libipc* common library functions. See *tuxshell*.

J

Julian date

Increasing count of the number of days since an arbitrary starting date.

L

LAN

Local Area Network.

launch

Initiate, spawn, execute, or call a software program or analysis tool.

LMID

Logical machine identifier: the logical reference to a machine used by a Tuxedo application. LMIDs can be descriptive, but they should not be the same as the UNIX hostname of the machine.

M

Map

Application for displaying S/H/I events, stations, and other information on geographical maps.

Master (machine)

Machine that is designated to be the controller of a DACS (Tuxedo) application. In the IDC application the customary logical machine identifier (LMID) of the Master is THOST.

message interval

Entry in a Tuxedo queue within the qspace referring to rows in the **interval** or **request** database tables. The DACS programs ensure that **interval** tables and qspace remain in synchronization at all times.

message queue

Repository for data intervals that cannot be processed immediately. Queues contain references to the data while the data remains on disk.

N

NFS

Network File System (Sun Microsystems). Protocol that enables clients to mount remote directories onto their own local filesystems.

O

online

Logged onto a network or having unspecified access to the Internet.

ORACLE

Vendor of the database management system used at the PIDC and IDC.

P

parameter (par) file

ASCII file containing values for parameters of a program. Par files are used to replace command line arguments. The files are formatted as a list of [*token* = *value*] strings.

partitioned

State in which a machine can no longer be accessed from other DACS machines via IPC resources *BRIDGE* and *BBL*.

PIDC

Prototype International Data Centre.

pipeline

1) Flow of data at the IDC from the receipt of communications to the final automated processed data before analyst review. 2) Sequence of IDC processes controlled by the DACS that either produce a specific product (such as a Standard Event List) or perform a general task (such as station processing).

PS

Processing server.

Q

qspace

Set of message queues grouped under a logical name. The IDC application has a primary and a backup qspace. The primary qspace customarily resides on the machine with logical reference (LMID) QHOST.

R

real time

Actual time during which something takes place.

run

(1) Single, usually continuous, execution of a computer program. (2) To execute a computer program.

S

SAIC

Science Applications International Corporation.

Scheme

Dialect of the Lisp programming language that is used to configure some IDC software.

script

Small executable program, written with UNIX and other related commands, that does not need to be compiled.

SEL1

Standard Event List 1; S/H/I bulletin created by total automatic analysis of continuous timeseries data. Typically, the list runs one hour behind real time.

SEL2

Standard Event List 2; S/H/I bulletin created by totally automatic analysis of both continuous data and segments of data specifically down-loaded from stations of the auxiliary seismic network. Typically, the list runs five hours behind real time.

SEL3

Standard Event List 3; S/H/I bulletin created by totally automatic analysis of both continuous data and segments of data specifically down-loaded from stations of the auxiliary seismic network. Typically, the list runs 12 hours behind real time.

server

Software module that accepts requests from clients and other servers and returns replies.

server (group)

Set of servers that have been assigned a common GROUPNO parameter in the `ubbconfig` file. All servers in one server group must run on the same logical machine (LMID). Servers in a group often advertise equivalent or logically related services.

service

Action performed by an application server. The server is said to be advertising that service. A server may advertise several services (multiple personalities), and several servers may advertise the same service (replicated servers).

shutdown

Action of terminating a server process as a memory-resident task. Shutting down the whole application is equivalent to terminating all specified server processes (admin servers first, application servers second) in the reverse order that they were booted.

Solaris

Name of the operating system used on Sun Microsystems hardware.

SRVID

Server identifier: integer between 1 and 29999 uniquely referring to a particular server. The SRVID is used in the `ubbconfig` file and with Tuxedo administrative utilities to refer to this server.

StaPro

Station Processing application for S/H/I data.

station

Collection of one or more monitoring instruments. Stations can have either one sensor location (for example, BGCA) or a spatially distributed array of sensors (for example, ASAR).

subsystem

Secondary or subordinate system within the larger system.

T

TI

Class of DACS servers that form time intervals by station sensor (for example, *tis_server*).

TMS

Transaction manager server.

transaction

Set of operations that is treated as a unit. If one of the operations fails, the whole transaction is considered failed and the system is "rolled back" to its pre-transaction processing state.

Tuxedo

Transactions for UNIX Extended for Distributed Operations.

tuxpad

DACS client that provides a graphical user interface for common Tuxedo administrative services.

tuxshell

Process in the Distributed Processing CSCI used to execute and manage applications. See IPC.

U

ubbconfig file

Human readable file containing all of the Tuxedo configuration information for a single DACS application.

UID

User identifier.

UNIX

Trade name of the operating system used by the Sun workstations.

V

version

Initial release or re-release of a computer software component.

W

waveform

Time-domain signal data from a sensor (the voltage output) where the voltage has been converted to a digital count (which is monotonic with the amplitude of the stimulus to which the sensor responds).

Web

World Wide Web; a graphics-intensive environment running on top of the Internet.

WorkFlow

Software that displays the progress of automated processing systems.

workstation

High-end, powerful desktop computer preferred for graphics and usually networked.

Index

A

admin server vii, 42
affiliation 27, 121, 122
application instances 5
application server vii, 43
    TMQFORWARD 44
    TMQUEUE 44
    TMS 43
    TMS_QM 43
    TMSYSEVT 44
    TMUSREVT 44
Automatic Processing 5
    conceptual data flow 14
    utilities 32
availability management requirements 128
    traceability 148

B

backup (component) viii
backup concept 23
BBL 42
birdie 100
    control 109
    error states 109
    I/O 105
    interfaces 109
boot viii
BRIDGE 19, 42
BSBRIDGE 42
bulletin board 42

C

capacity mapping 24
catchup capability 24
client viii
conventions
    data flow symbols v
    entity-relationship symbols vi
    typographical vii
CSCI external interface requirements 137
    traceability 161
CSCI internal data requirements 142
    traceability 169

D

DACS
    filesystem use 20
    interface with other IDC systems 34
    machines viii
    operational modes 127
    operator interface 35
data flow symbols v
data monitors viii
    ticron_server 63, 64, 67
    tiseg_server 61
data monitor servers 54
DBBL 42
dbserver 31, 32, 51, 89, 91
    control 92
    error states 93
    I/O 91
    interfaces 92

dequeue viii
distinguished bulletin board 43
distributed processing 8, 23
distribution objectives 24
dman 100
control 109
error states 109
I/O 105
interfaces 109

E

enqueue viii
entity-relationship symbols vi

F

forwarding agent viii, 23
functional requirements 128
traceability 148, 150, 153, 156, 158

G

generalized processing server (*tuxshell*) viii
general requirements 126
traceability 144

H

hardware requirements 11
host 21

I

instance viii
Interactive Processing 6, 32
conceptual data flow 16
interval ix
interval 27, 31, 59, 121, 122
interval_router 30, 32, 49, 90
control 92
error states 93
I/O 91
interfaces 92
IPC resources 45

L

lastid 27, 121, 122
libgdi 119
libipc 19, 100
control 109
error states 109
I/O 105
interfaces 109
libraries, global 18
listener daemons (*tlisten* and *tagent*) 38
LMID ix
load balancing 24
load limitation 24
log files 20

M

Master ix
message ix
message passing requirements 129
traceability 150
message queue ix, 21, 45
middleware 7
minimization of network traffic 24
msg_window 110
control 118
error states 119
I/O 115
interfaces 118

N

network processing 63, 64, 67

O

operate_admin 110
control 118
error states 119
I/O 115
interfaces 118
operational modes 127

P

partitioned ix
pipeline
description 25
schematic 26
ProcessInterval 31, 93

Q

qinfo 110
control 118
error states 119
I/O 115
interfaces 118
qmadmin 46
qspace ix, 46
queues 46
queue space 46

R

recycler_router 90
recycler_server 32, 51
control 92
error states 93
I/O 92
interfaces 92
reliability requirements 134
traceability 158
request 28, 31, 68, 75, 121, 122
requirements
COTS software 11
CSCI external interface 137
CSCI internal data 142
functional 128
general 126
hardware 11
system 142
requirements traceability 144
rollback 22

S

schedclient 31, 49, 77
control 81
error states 82
I/O 78
interfaces 82
schedule_it 110
control 118
error states 119
I/O 115
interfaces 118
scheduler 30, 49, 77
control 81
error states 82
I/O 78
interfaces 82
semaphores 45
SendMessage 31, 93
server 21
server group ix
service ix, 21
shared memory 45
single-point-of-failure 24
software requirements 11
SRVID x

system monitoring requirements 133
traceability 156
system requirements 142
traceability 169

T

tagent 42
technical terms vii
ticron_server 30, 54, 63, 64, 67
I/O 71
timestamp 28, 75, 121, 123
tin_server 30, 54
I/O 72
tis_server 30, 49, 57
I/O 69
tis_server,tiseg_server 54
tiseg_server 30, 61
I/O 70
tlisten 38
tmadmin 46
tmloadcf 46
TMQFORWARD 23, 31, 44, 51
TMQUEUE 44
TMS 43
TMS_QM 43
TMSYSEVT 44
tmunloadcf 46
TMUSREVT 44
transaction x
description 22
transactional resource managers 9
transaction logs 45
`tuxconfig` 45
tuxpad x, 32, 49, 110
control 118
error states 119
I/O 115
interfaces 118
tuxshell 31, 83
control 88
error states 88
I/O 86
interfaces 88
typographical conventions vii

U

`ubbconfig` 20, 45
file x
user logs 45
utility programs (*tmadmin*, *qmadmin*) 46

W

WaveGet_server 30, 54
I/O 74
wfdisc 28, 59, 123
WorkFlow 31, 49, 93
control 99
error states 100
I/O 96
interfaces 99
workflow management requirements 131
traceability 153